Programmation Procédurale en Langage C

Cours et Exercices Pratiques — S3-PRC
École Nationale d'Ingénieurs de Brest (ENIB)

Fabrice Harrouet, version du 26 janvier 2018

Table des matières

Avant-propos **9**

1. C01_Basics : Éléments de base du langage **11**

1.1. Pourquoi le langage C ? ... 11

1.2. Un programme est un ensemble de fonctions ... 12
1.2.1. Fabrication d'un programme ... 13
1.2.2. La fonction principale ... 13
1.2.3. Le prototype d'une fonction ... 14
1.2.4. Les fonctions externes ... 15

1.3. Typage des données ... 16
1.3.1. Déclaration de variables ... 16
1.3.2. Quelques types usuels ... 17
1.3.3. Conversion de type ... 20
1.3.4. Portée et classe de stockage des variables ... 21

1.4. Les règles d'écriture du langage ... 22
1.4.1. Le préprocesseur ... 22
1.4.2. La mise en forme ... 24
1.4.3. Les identificateurs ... 25
1.4.4. Les blocs, les instructions et les expressions ... 25
1.4.5. Les instructions de contrôle ... 28

1.5. Résumé ... 35

2. L01_Build : Organisation et fabrication d'un programme **37**

2.1. Un premier programme en langage C ... 37
2.1.1. Rédaction du code source ... 37
2.1.2. Compilation du programme ... 38
2.1.3. Exécution du programme ... 40

2.2. Vers une approche modulaire du développement ... 41
2.2.1. Référence à un module de l'application ... 41
2.2.2. Ajout d'un module à l'application ... 42

2.3. Démarche de compilation séparée ... 43
2.3.1. Modification d'un module ... 43

2.3.2. Digression : analyse d'un plantage au débogueur 44
2.3.3. Cohérence entre déclaration et définition 45
2.3.4. Modules interdépendants 47
2.3.5. Digression : analyse de l'exécution au débogueur 49
2.4. Automatisation de la construction 50
2.4.1. Un fichier makefile très spécifique 50
2.4.2. Généralisation du fichier makefile 52
2.5. Résumé 54

3. C02_Pointers : Pointeurs et tableaux 59

3.1. Les passages de paramètres 59
3.1.1. Passage par valeur 59
3.1.2. Passage par adresse 60
3.2. Les variables de type pointeur 61
3.2.1. Déclaration de pointeurs 61
3.2.2. Utilisation d'un pointeur 62
3.2.3. Un pointeur comme type de retour 62
3.3. Les variables de type tableau 63
3.3.1. Déclaration et utilisation d'un tableau 63
3.3.2. Initialisation d'un tableau 64
3.3.3. Précautions pour l'usage des tableaux et des pointeurs 65
3.4. Opérations sur les pointeurs 68
3.4.1. Taille d'une variable de type pointeur 68
3.4.2. Arithmétique des pointeurs 69
3.4.3. Valeur booléenne d'un pointeur 70
3.4.4. Conversion du type d'un pointeur 71
3.4.5. Le qualificatif const 71
3.4.6. Pointeur sur pointeur 72
3.5. Résumé 73

4. L02_Types : Variété de types 75

4.1. Mise en place du programme 75
4.2. La taille des types 76
4.2.1. L'opérateur sizeof 76
4.2.2. Comparaison de la taille des types 76
4.2.3. Les entiers de taille déterminée 77
4.3. Dépassement de capacité 77
4.3.1. Conversion vers un type insuffisamment large 77
4.3.2. Incrémentation excessive 78
4.3.3. Dépassement dans un calcul arithmétique 79
4.3.4. Promotion automatique des petits entiers 80
4.4. Les limites des réels 81

4.4.1. L'arrondi de la représentation réelle 81
4.4.2. Les ordres de grandeur des valeurs réelles 82
4.4.3. Application au développement en série 83
4.4.4. Digression : étude de l'évolution des variables au débogueur 84
4.5. Résumé 85

5. C03_Strings : Chaînes de caractères **91**

5.1. Séquences de caractères 91
5.1.1. Le type des caractères 91
5.1.2. Les caractères littéraux 92
5.1.3. Les catégories de caractères 92
5.1.4. Le zéro terminal 93
5.2. Initialisation des chaînes de caractères 94
5.2.1. Les chaînes littérales 94
5.2.2. Affectation à un pointeur 94
5.3. Les fonctions de manipulation de chaînes 95
5.3.1. Longueur d'une chaîne 95
5.3.2. Comparaison de chaînes 96
5.3.3. Recherche dans une chaîne 97
5.3.4. Recopie d'une chaîne 97
5.4. La ligne de commande 98
5.5. Résumé 99

6. L03_Alloc : Allocation dynamique **101**

6.1. Mise en place du programme 101
6.2. Limite de la pile d'exécution 102
6.3. Allocation dynamique de mémoire 104
6.3.1. Prise en main 104
6.3.2. Digression : contrôle de l'échec de l'allocation 105
6.3.3. Intégration au module 106
6.4. Ré-allocation de la mémoire dynamique 106
6.5. Allocation spéculative 108
6.6. Résumé 109

7. C04_Struct : Structures de données **117**

7.1. Un nouveau type de données 117
7.1.1. Définition d'une structure 117
7.1.2. Accès aux membres d'une structure 118
7.1.3. Initialisation d'une structure 119
7.1.4. Copie d'une structure 119
7.1.5. Disposition des membres d'une structure 120

7.2. Usage effectif des structures 121
7.2.1. Un type simple : un peu plus qu'un type de base 122
7.2.2. Un type élaboré : gestion de ressources 123
7.2.3. Optimisation de l'interface 124
7.3. Résumé 127

8. L04_IO : Entrées-sorties **129**

8.1. Mise en place du programme 129
8.2. Formatage de texte 129
8.3. Sortie standard et sortie d'erreurs standard 130
8.4. Écriture de texte dans un fichier 131
8.5. Sauvegarde d'une image dans un format textuel 132
8.6. Entrée standard 133
8.7. Lecture d'une image dans un format textuel 134
8.8. Entrées-sorties dans un format binaire 135
8.8.1. Lecture et écriture de données binaires dans un fichier 135
8.8.2. Sauvegarde d'une image dans un format binaire 136
8.8.3. Lecture d'une image dans un format binaire 137
8.9. Analyse de la ligne de commande 137
8.10. Résumé 138

9. L05_Bitwise : Opérations bit-à-bit **147**

9.1. Mise en place du programme 147
9.2. Les opérations de décalage 148
9.3. Les combinaisons bit-à-bit 149
9.4. Test de bits et forçage à 1 150
9.5. Complémentation et forçage de bits à 0 151
9.6. Opérations bit-à-bit sur les entiers signés 151
9.7. Résumé 153

10. L06_Maths : Opérations mathématiques **159**

10.1. Mise en place du programme 159
10.2. La bibliothèque mathématique 159
10.2.1. Mise en œuvre de la bibliothèque mathématique 160
10.2.2. Détection des valeurs extrêmes 161
10.2.3. Arrondi d'un réel vers un entier 161
10.2.4. Les valeurs réelles anormales 163

10.3. Génération pseudo-aléatoire 164
10.3.1. Obtention de valeurs pseudo-aléatoires 164
10.3.2. Choix de la plage de valeurs pseudo-aléatoires 165
10.3.3. Distributions de valeurs pseudo-aléatoires réelles 165
10.4. Résumé 166

11. L07_FnctPtr : Pointeurs sur fonctions **173**
11.1. Mise en place du programme 173
11.2. Déclaration et utilisation d'un pointeur sur fonctions 174
11.3. Généralisation d'un traitement 175
11.4. Paramétrage des traitements génériques 176
11.5. Résumé 178

12. L08_Perfs : Optimisation des performances **183**
12.1. Mise au point ou optimisation 183
12.1.1. Permettre l'usage d'un débogueur 183
12.1.2. Détecter au plus tôt les incohérences dans l'application 184
12.1.3. Instrumenter le code pour détecter les maladresses 184
12.1.4. Compiler pour l'optimisation 185
12.2. Recommandations pour l'optimisation 185
12.2.1. Éliminer les répétitions inutiles 186
12.2.2. La vision limitée du compilateur 187
12.2.3. Expliciter nos intentions 188
12.2.4. Préférer les entiers signés aux entiers non-signés 189
12.3. Résumé 190

Glossaire **195**

Avant-propos

Ce document contient à la fois des cours et des sujets de séances pratiques pour découvrir le langage C dans le cadre de la matière S3-PRC de l'École Nationale d'Ingénieurs de Brest (ENIB). Cet enseignement n'a pas vocation à être exhaustif sur tous les détails de ce langage. Il s'agit plutôt de fournir les éléments nécessaires à une maîtrise suffisante pour être en mesure :

- de comprendre l'essentiel des réalisations existantes,
- de les modifier et d'en produire de nouvelles,
- de disposer d'éléments de compréhension pour exploiter une documentation détaillée.

Ce support a été rédigé avec à l'esprit une démarche d'apprentissage progressive au fil du semestre. Il doit être étudié du début vers la fin car chaque chapitre ne dépend de ce qui précède (bien que des renvois permettent de renforcer les liens entre les notions). Il ne s'agit en aucun cas d'un manuel de référence regroupant par catégories les fonctionnalités du langage. Il existe de nombreux ouvrages dédiés à ce propos ; ce document fait notamment de très nombreuses références au site :

(http://en.cppreference.com/w/c)

Cet enseignement est au contraire organisé par thèmes et vise principalement à porter l'accent sur les usages pratiques des fonctionnalités du langage en apportant les recommandations qui sont jugées pertinentes au regard de l'expérience des enseignants et du domaine d'application de ce langage.

L'utilisation directe de cet enseignement dans le contexte de l'ENIB concerne principalement les matières S4-PRC (extension au langage C++), S4-CEL (construction électronique), S5-MIP et S6-MIP (microprocesseurs) ; toutefois d'autres matières, unités d'enseignement et projets en font régulièrement usage.

Les chapitres de ce document sont respectivement préfixés par C□□ pour les cours, c'est à dire l'acquisition de connaissances académiques, et L□□ pour les labos, c'est à dire des exercices pratiques visant à découvrir des notions par l'expérience. Les cours peuvent facilement être travaillés en autonomie puisqu'il ne s'agit que de lecture attentive.

Les exercices pratiques nécessitent quant à eux de disposer d'un poste de travail sur lequel les outils de développement en langage C ont été correctement installés. C'est le cas sur les postes de l'ENIB qui sont équipés du système d'exploitation Linux. Pour une installation sur un poste personnel, des consignes sont données sur le site dédié à cet enseignement.

(http://www.enib.fr/~harrouet/s3prc/)

Ce site propose notamment des questionnaires et des exercices d'entraînement aux épreuves pour accompagner les cours et les labos.

Bien entendu, au-delà du travail personnel, des enseignants vous accompagneront pendant les séances prévues à l'emploi du temps de l'ENIB afin de répondre à vos questions et vous guider dans votre démarche d'apprentissage.

L'organisation des examens reprendra la démarche que vous suivez depuis la matière S1-ALR : vous passez les épreuves au moment au vous vous sentez prêt. Le site dédié à cette matière indique les modalités de validation.

1. C01_Basics : Éléments de base du langage

Après vous avoir présenté la place du langage C parmi l'ensemble des outils disponibles en matière de développement informatique, l'objectif principal de ce premier cours est de vous apporter les éléments généraux permettant la réalisation de programmes simples avec ce langage. Les caractéristiques plus avancées de ce langage ne seront abordés que dans les cours et les exercices pratiques ultérieurs. Tous les détails du langage pourront être retrouvés sur un site de référence auquel ce document renvoie très régulièrement :

(http://en.cppreference.com/w/c)

Les notions d'algorithmique et de développement informatique en langage Python, respectivement étudiées dans les matières S1-ALR et S2-IPI de l'ENIB, sont supposées assimilées et servent ici de prérequis.

1.1. Pourquoi le langage C ?

Le langage C est apparu au début des années 1970. À l'époque, le développement sur les (rares) machines informatiques nécessitait l'emploi du langage d'assemblage[1] spécifique à chacune d'elles. Le portage d'un programme, voire d'un système d'exploitation, depuis un modèle de machine vers un autre nécessitait alors une réécriture totale. En autorisant l'expression des constructions algorithmiques usuelles (variables, boucles, fonctions...) dans un langage de plus haut niveau (plus facile à lire pour un être humain) qui pouvait être traduit (compilé) dans le langage spécifique à la machine sous-jacente, tout en autorisant l'accès aux ressources de bas niveau de cette machine (les périphériques et la mémoire notamment), le langage C pouvait être considéré comme un langage d'assemblage portable. Il s'est imposé comme le langage de choix pour la réalisation du premier système d'exploitation portable largement déployé : le système UNIX.

(https://en.wikipedia.org/wiki/C_(programming_language))

Au fil de l'histoire, la norme décrivant ce langage a subi plusieurs évolutions. Même si aujourd'hui nous en sommes à la version C11 (introduite en 2011), la version du langage C qui est encore à l'heure actuelle la plus largement disponible et utilisée repose sur la norme C99 (introduite en 1999). C'est sur celle-ci que nous nous appuierons dans le cadre de l'apprentissage de ce langage[2].

Pendant très longtemps le langage C est resté le langage industriel prédominant pour tout développement conséquent. Seulement, à partir du milieu des années 1990 le foisonnement de nouvelles technologies et de langages associés offrait aux programmeurs un large éventail d'outils pour accomplir plus confortablement les tâches de chaque domaine applicatif spécifique. Pour accentuer ce phénomène, jusqu'au début des années 2000 la montée en fréquence de chaque nouvelle génération de processeurs était telle qu'il était plus rentable économiquement de miser sur l'achat de machines modernes et puissantes, que de payer des spécialistes pour optimiser l'efficacité des programmes. Cela a conduit à un moindre besoin en programmeurs maîtrisant le fonctionnement de bas niveau des machines informatiques, au

1 Le fonctionnement et la programmation des microprocesseurs seront étudiés ultérieurement dans le cursus de votre formation à l'ENIB (matières S5-MIP et S6-MIP).

2 Les programmeurs qui veulent accéder à des fonctionnalités avancées se tournent désormais plus volontiers vers le langage C++17 comme nous le verrons dans la matière S4-PRC de l'ENIB.

profit d'une forte demande en développeurs capable de produire rapidement du code moins efficace avec des outils de haut niveau. C'est là toute l'ironie du sort : le langage C, qui à l'origine était vu comme un langage de haut-niveau, était devenu un langage de bas niveau !

Toutefois, la donne a une nouvelle fois changé après le début des années 2000. La prétendue inexorable montée en fréquence des nouveaux processeurs a finalement atteint une limite unanimement considérée comme infranchissable (ce n'est pas qu'une question de temps ni de technique). La montée en puissance repose désormais en grande partie sur la multiplication des unités de calcul (processeurs multi-cœurs et machines multi-processeurs) ce qui nécessite à nouveau un contrôle fin des détails d'exécution des programmes pour en tirer pleinement parti ; le langage C représente l'outil de prédilection pour offrir une telle maîtrise. À cette nouvelle tendance s'ajoute le fait que la problématique de l'économie de l'énergie, et à plus forte raison dans les domaines de l'informatique mobile et embarquée où la question de l'autonomie est déterminante, encourage à exploiter au mieux les capacités de calcul des processeurs. Il n'est désormais plus envisageable d'utiliser un processeur démesurément puissant, et donc énergivore, là où un processeur plus économe en énergie ferait l'affaire à condition d'optimiser les traitements qu'on lui confie. Tout gain en efficacité de calcul se traduit par une économie d'énergie, donc un gain en autonomie ; de manière complémentaire, dans une enveloppe énergétique et thermique donnée les traitements fournissent au bout du compte plus de service si on les optimise. Une nouvelle fois, le langage C s'avère parfaitement adapté à ces enjeux[3].

Malgré son age avancé, le langage C reste alors toujours un outil incontournable pour les ingénieurs. Même si à l'heure actuelle il est assez rare de l'utiliser pour développer une application complète, il est très fréquent d'y avoir recours pour réaliser des bibliothèques de fonctionnalités qui sont invoquées depuis des langages de plus haut niveau lorsque ceux-ci souffrent de limitations en performances. C'est justement une compétence importante de l'ingénieur que de savoir déterminer et isoler les parties d'une application qui bénéficieraient d'une telle reformulation en langage C. Voici quelques cas typiques d'utilisation :

- besoin d'accéder à des ressources de bas niveau, la programmation de pilotes de périphériques (*drivers*) par exemple[4],
- ressources matérielles limitées en calcul et/ou en mémoire (domaine de l'informatique mobile et embarquée[5]) ; d'où l'intérêt d'un langage qui ne gaspille pas de ressources,
- applications dans des langages de plus haut niveau contenant néanmoins des traitements très calculatoires qui peuvent être longs ; il est alors nécessaire d'isoler ces traitements pour les réécrire avec un langage plus efficace et rendre l'application plus réactive et confortable,
- très longs calculs, simulations biologiques ou physiques pouvant durer plusieurs jours par exemple, sur du matériel non limité en ressources ; la moindre optimisation peut faire gagner un temps et donc une énergie considérables.

Pour terminer cette introduction, soulignons le fait que la syntaxe du langage C (que nous nous proposons de découvrir dans ce premier cours) a été reprise dans une multitude d'autres langages, même de haut niveau. En particulier, le langage C++ (à plus forte raison dans sa version moderne) conserve au premier plan le soucis de la maîtrise de l'optimisation et des détails d'exécution des programmes et reste en grande partie compatible avec le langage C dont il est directement issu.

1.2. Un programme est un ensemble de fonctions

Nous décrivons ici la forme générale d'un programme en langage C sans rentrer dans les détails de l'expression des algorithmes.

---

3 La partie introductive de l'exposé *« Efficiency with Algorithms Performance with Data Structures »*, par Chandler Carruth à la conférence CppCon 2014, est très explicite à ce sujet (vidéo facilement accessible en ligne).

4 L'enseignement autour des microprocesseurs à l'ENIB (matières S4-CEL, S5-MIP et S6-MIP) s'appuie en grande partie sur le langage C.

5 Même remarque.

1.2.1. Fabrication d'un programme

Rédiger un programme en langage C revient à définir, dans un ou plusieurs fichiers textuels, un ensemble de fonctions. Contrairement à un programme Python, pour lequel il suffit d'invoquer l'interpréteur en lui indiquant le fichier de code source, un programme en langage C ne peut pas être directement exécuté. Il faut passer par une phase intermédiaire qui traduit ce code source en un format exécutable que le système d'exploitation et la machine informatique comprennent : il s'agit de la ***compilation*** (terme de vocabulaire à connaître).

L'outil qui analyse le code source pour réaliser cette traduction s'appelle un ***compilateur*** (vocabulaire). Son rôle premier consiste à vérifier que le code source répond bien aux règles d'écriture du langage (sa grammaire et sa sémantique) et à signaler les erreurs éventuelles. Ce qui le différencie fondamentalement d'un interpréteur (comme celui de Python) c'est qu'il vérifie à l'avance la cohérence des instructions invoquées dans le code source en s'appuyant notamment sur les types des données manipulées (l'addition d'entiers n'a rien à voir avec celle de réels et encore moins avec celle de chaînes de caractères). Le code exécutable résultant ne contient alors que les instructions utiles au calcul ; toutes les vérifications qui consistent à contrôler le type des données manipulées et la cohérence des opérations qu'on y applique n'ont pas lieu durant l'exécution du programme mais bien avant, une fois pour toutes, lors de sa compilation. De plus, une fois que le compilateur a déterminé l'ensemble des instructions utiles qu'il doit générer, il se permet de les simplifier et de les réordonner afin que l'exécution du traitement soit la plus rapide possible tout en restant conforme aux algorithmes exprimés dans le code source. Cette phase s'appelle l'***optimisation*** (vocabulaire). Toutes ces raisons font qu'au bout du compte un langage compilé comme le C produit généralement des programmes plus efficaces qu'un langage interprété comme Python.

L'organisation des fichiers constituant le code source et la démarche de mise en œuvre d'un tel outil seront étudiées en exercice pratique (chapitre 2).

1.2.2. La fonction principale

Vos premiers pas en algorithmique vous ont invité à saisir directement les premières instructions en langage Python en dehors de tout contexte. Ce n'est que plus tard que vous avez découvert et utilisé la notion de fonction, et encore bien après que vous avez été incité à décrire globalement un projet comme un ensemble de fonctions interdépendantes. Le point d'entrée de votre programme était alors à nouveau une instruction isolée qui invoquait une fonction particulière que vous considériez comme la fonction principale du programme : celle depuis laquelle toutes les autres sont invoquées, directement ou indirectement.

En langage C, il n'est pas question d'écrire des instructions en dehors de tout contexte ; toute instruction fait forcément partie d'une fonction. Par conséquent, ce langage impose la présence d'une fonction particulière qui sera considérée comme la ***fonction principale*** (vocabulaire) du programme : il s'agit de la fonction *`main()`*[6]. Elle est automatiquement appelée quand le programme démarre ; lorsque cette fonction se termine, le programme est également terminé.

Voici comment se présente la fonction principale d'un programme en langage C :

```
int        // cette fonction doit renvoyer un entier
main(void) // elle s'appelle main et n'attend aucun paramètre
{

// cette paire d'accolades matérialise le corps de la fonction,
// c'est ici que figureront les variables et les instructions

return 0;  // voici l'entier que cette fonction doit renvoyer
}
```

6 Un ensemble de fonctions qui ne fournit pas de fonction *`main()`* ne constitue pas un programme. En revanche, il peut s'agir d'une bibliothèque de fonctions qui pourront ultérieurement être incorporées dans un programme.

Les portions de lignes qui contiennent le symbole `//` (*double-slash*) sont des commentaires qui permettent d'ignorer le texte qui s'étend de ce symbole jusqu'à à la fin de la ligne (similaire à `#` que vous connaissez déjà en langage Python). La construction de la forme `int main(void)` indique qu'il s'agit de la fonction nommée `main` qui n'attend aucun paramètre (signifié par le mot-clef `void`, contrairement à Python ou C++ où des parenthèses vides suffisent), et qui renvoie comme résultat un entier (mot-clef `int`, présenté prochainement). La paire de symboles `{ }` (accolades) qui suit matérialise le corps de la fonction c'est à dire l'algorithme que doit exécuter cette fonction lorsqu'elle est appelée ; il ne s'agit dans cet exemple simpliste que de fournir une valeur entière (`0`) que cette fonction doit renvoyer.

Nous n'avons pas le choix quant au fait que la fonction `main()` n'attend pas de paramètre (à une variante près qui sera vue dans le cours du chapitre 5). Même si pour l'instant nous ne faisons apparemment aucun usage de l'entier que nous lui faisons renvoyer (puisque le programme se termine à la sortie de cette fonction), celui-ci est obligatoire (son usage sera constaté dans l'exercice pratique du chapitre 2).

(http://en.cppreference.com/w/c/language/main_function)

1.2.3. Le prototype d'une fonction

Bien que la fonction `main()` présentée précédemment soit particulière (la fonction principale du programme), toutes les fonctions d'un programme ont une forme semblable. En particulier, elles ont en commun le fait d'être désignées par un nom, d'être invoquées en recevant des paramètres et de pouvoir fournir un résultat. La description de ces propriétés s'appelle le ***prototype*** (vocabulaire) de la fonction. Cette notion a un rôle prépondérant en langage C.

En effet, lorsque dans un algorithme il est effectué un appel à une fonction, le compilateur qui analyse cette instruction s'assure que les paramètres effectifs transmis sont bien conformes aux paramètres formels attendus et qu'il est fait bon usage de la valeur renvoyée. Veuillez remarquer qu'à ce stade, les vérifications sont indépendantes des détails de réalisation de l'algorithme invoqué ; seul le prototype est nécessaire. Puisque le compilateur de langage C analyse le code source de haut en bas, il doit avoir rencontré la formulation du prototype d'une fonction avant toute instruction d'appel à cette même fonction.

C'est le cas sur cet exemple :

```
int                       // cette fonction doit renvoyer un entier
axpy(int a, int x, int y) // elle s'appelle axpy et attend trois paramètres entiers
{
return a*x+y;             // elle combine ses paramètres pour produire son résultat
}

int
main(void)
{                         // la fonction axpy() est invoquée avec les trois entiers attendus,
return 10*axpy(2,5,7);    // son résultat est exploité pour produire le résultat du programme
}
```

dans lequel la fonction `axpy()` est bien entièrement définie avant son appel depuis la fonction `main()`. Remarquez que dans un prototype il y a un et un seul type de retour ; ici il s'agit du type `int` mais si aucun résultat ne devait être transmis nous aurions écrit le mot-clef `void` à la place. Les paramètres d'une fonction sont séparés deux à deux par une virgule et sont constitués chacun d'un type et d'un nom ; ici la fonction `axpy()` attend trois paramètres mais la fonction `main()` n'en attend aucun et le signifie par l'usage du mot-clef `void`.

Ce même programme aurait pu être décrit par cette formulation tout aussi valide :

```
int                         // cette fonction doit renvoyer un entier
axpy(int a, int x, int y); // elle s'appelle axpy et attend trois paramètres entiers

int
main(void)
{                          // la fonction axpy() est invoquée avec les trois entiers attendus,
return 10*axpy(2,5,7);     // son résultat est exploité pour produire le résultat du programme
}

int                        // la définition de la fonction axpy() a bien le même prototype
axpy(int a, int x, int y) // que sa déclaration préalable
{
return a*x+y;              // elle combine ses paramètres pour produire son résultat
}
```

dans laquelle la fonction *axpy()* est à nouveau entièrement définie mais cette fois-ci après la fonction main(). Pour que le compilateur accepte d'analyser l'appel à *axpy()* depuis *main()*, il nous a été nécessaire de déclarer au préalable le prototype de *axpy()*.

Cette formulation qui consiste à écrire le prototype d'une fonction suivi du symbole *;* (point-virgule) s'appelle une ***déclaration*** (vocabulaire). Une déclaration constitue en quelque sorte une promesse qui est faite au compilateur pour lui indiquer que quelque part existe une fonction ayant ce nom et correspondant à ce prototype. Une fois que cette promesse lui est faite, le compilateur peut légitimement valider et traduire des instructions qui cherchent à appeler cette fonction si elles font un bon usage de ses paramètres et de son résultat.

La formulation qui consiste à écrire le prototype d'une fonction suivi du bloc de code (les accolades) qui constitue son algorithme s'appelle une ***définition*** (vocabulaire). Il est absolument indispensable que le prototype de la définition d'une fonction soit identique à celui de sa déclaration, sinon le compilateur signale une erreur. Bien que ce ne soit pas recommandé, les noms choisis pour les paramètres peuvent éventuellement différer (mais pas leurs types) et peuvent même être omis[7]. Une telle définition de la fonction doit bien entendu figurer quelque part dans le programme puisqu'il faudra bien exécuter ses instructions quand cette fonction sera appelée ; la déclaration n'est pas suffisante dans l'absolu.

Il est très important de savoir distinguer la déclaration de la définition d'une fonction ; la définition fournit du code, la déclaration non ! Cette distinction révèle son intérêt ci-dessous.

1.2.4. Les fonctions externes

Les quelques exemples simplistes présentés jusqu'alors fournissent en un unique fichier l'intégralité du code source du programme. Cependant, tout projet raisonnable est fragmenté en un ensemble de modules et de bibliothèques. C'est dans ces conditions que les déclarations de fonctions prennent toute leur utilité ; elles déclarent les fonctions qui sont définies dans un autre module ou une autre bibliothèque.

Plutôt que de multiples rédactions explicites de ces déclarations dans chaque fichier de code source qui le nécessite (ce qui serait source d'erreurs de saisie), la pratique unanimement adoptée consiste à les rédiger une fois pour toutes dans un ***fichier d'en-tête*** (vocabulaire) dédié à cet effet. Ce fichier est alors incorporé, par une directive d'inclusion qui sera présentée ultérieurement, dans chaque fichier de code source qui nécessite ces déclarations.

Cet exemple emblématique d'introduction au langage C :

7 Un programmeur qui lit un prototype doit comprendre le rôle de la fonction et de ses paramètres. Omettre ou modifier les noms des paramètres dans la déclaration ne fait qu'ajouter de la confusion.

```
#include <stdio.h>

int
main(void)
{
printf("hello, world\n");
return 0;
}
```

réclame l'inclusion du fichier d'en-tête *stdio.h*. Ce dernier contient des déclarations de fonctions qui sont déjà définies dans la ***bibliothèque standard*** (vocabulaire) du langage C. Cette bibliothèque fournit un ensemble de fonctionnalités déjà compilées et livrées telles quelles lors de l'installation du compilateur sur le système d'exploitation. Tout programme peut y faire appel à condition d'en connaître les prototypes ; le fichier d'***en-tête standard*** (vocabulaire) *stdio.h* expose entre autres la déclaration de la fonction *printf()* qui est ici invoquée dans la fonction *main()*. Ce programme s'appuie donc sur la fonction externe *printf()*, définie dans la bibliothèque standard, pour afficher le texte littéral exprimé entre les symboles " (guillemets)[8].

Pour une expérimentation pratique autour de ce thème, un sujet d'exercice pratique (chapitre 2) est consacré à l'étude de l'organisation modulaire et à la compilation du code d'un programme.

1.3. Typage des données

Comme indiqué en 1.2.1, le travail du compilateur repose en grande partie sur l'analyse du type des données manipulées par le programme. Nous en avons déjà découvert l'usage lors de la description du prototype d'une fonction mais ces types interviennent également dans l'usage des variables.

1.3.1. Déclaration de variables

Vous connaissez déjà la notion de variable en algorithmique. Lorsque vous utilisiez le langage Python il suffisait d'affecter une valeur à une variable pour que cette dernière existe et soit utilisable dans la suite du code. Dans ces conditions, le type de l'information contenue dans une variable dépendait uniquement de la valeur utilisée pour cette affectation ; il était même envisageable en Python d'affecter une valeur d'un tout autre type à une variable existante.

En langage C il en est tout autrement : chaque variable a un type qui lui est attribué une fois pour toutes lors de sa déclaration. Cette variable ne peut alors contenir que des valeurs ayant le type choisi. C'est sur cette propriété que le compilateur de langage C s'appuie pour vérifier la cohérence des opérations et pour générer du code exécutable qui traite exactement du bon type de donnée.

Nous déclarons généralement les variables dans les fonctions, au plus près des algorithmes qui en ont besoin. Dans cette variante de la fonction *axpy()* utilisée dans nos exemples :

```
int
axpy(int a, int x, int y)
{
int myVar;     // déclaration d'une variable nommée myVar pouvant contenir une valeur entière
myVar=a*x;     // le produit de deux entiers peut être stocké dans une variable de type entier
myVar=myVar+y; // consultation et modification de cette variable
return myVar;  // la valeur de cette variable est du même type que le résultat
}
```

nous constatons que le nom d'une variable est précédé d'un type (ici *int* pour un entier) et suivi du symbole *;* (point-virgule) pour former sa déclaration. Une telle variable a une ***classe de stockage automatique*** (vocabulaire), c'est à dire qu'elle apparaît lorsque l'exécution atteint le bloc de code (paire d'accolades) dans lequel elle est déclarée et disparaît en en

8 Vous reconnaissez certainement le caractère spécial *\n* (backslash-n) qui matérialise un passage à la ligne.

sortant. De ce fait, ***lors de son apparition une variable a une valeur indéterminée*** qui dépend de l'état de la mémoire de la machine à l'endroit réservé pour son stockage : dans la ***pile d'exécution*** (vocabulaire)[9]. Elle peut alors accueillir, autant de fois que nécessaire, une valeur conforme à son type de déclaration afin de la restituer ultérieurement. Remarquons que les paramètres d'une fonction ne sont rien d'autres que des variables dont les valeurs ont été initialisées lors de l'appel. Il est important de noter que toutes ces variables ne sont visibles que dans la fonction à laquelle elles appartiennent ; des variables de même nom dans deux fonctions différentes sont complètement distinctes.

Il existe des variantes dans la façon de déclarer des variables ; ceci permet d'en déclarer plusieurs à la fois, ou encore de leur donner une valeur initiale explicite.

```
...
{
int v1;     // une seule variable de type entier de valeur indéterminée
int v2, v3; // deux variables de type entier de valeurs indéterminées
int v4=8;   // une seule variable de type entier valant initialement 8
int v5=10, v6, v7=v4+20, v8=2*v7, v9; // cinq variables de type entier, trois sont initialisées,
...                                   // deux autres ont une valeur indéterminée
}
```

Remarquez que lorsqu'une valeur initiale est spécifiée, elle peut dépendre de ce qui précède. Bien que ce ne soit pas imposé par le langage, il est recommandé de toujours donner une valeur initiale aux variables qu'on déclare. En effet, si on exprime des calculs qui dépendent d'une variable dont la valeur est encore indéterminée, cela donne une situation incohérente difficile à déboguer.

1.3.2. Quelques types usuels

Jusqu'alors, nous n'avons utilisé que le type `int` dans nos exemples. Bien évidemment le langage C propose une variété de types et offre même la possibilité d'en inventer de nouveaux. Nous présentons ici l'usage de seulement quelques-uns d'entre eux afin d'aborder l'écriture de programmes simples.

Tous les types de données doivent pouvoir être stockés dans la mémoire. La plus petite entité adressable (que l'on peut désigner pour y lire ou y écrire) dans la mémoire est communément appelée un ***byte*** (vocabulaire). Dans la pratique, bien que rien ne l'impose, il s'agit généralement d'un ***octet*** (vocabulaire), c'est à dire un ensemble de huit ***bits*** (*binary-digits*, vocabulaire), c'est pourquoi l'amalgame entre les termes *byte* et octet est très commun. Tout type de donnée est donc représenté dans la machine par un ou plusieurs octets.

Le type `int` : les entiers

Le type `int` permet de manipuler des valeurs entières. Les entiers sont représentés dans la machine informatique par une séquence de *bits* dont chacun représente une puissance de deux. De façon surprenante, le nombre d'octets constitutifs d'un `int` n'est pas spécifié en langage C. Chaque combinaison de matériel, système d'exploitation et compilateur peut décider de cette quantité, ce qui rend non portable tout programme qui suppose une taille précise pour ce type. Ce que nous savons de manière sûre, c'est qu'il tient au minimum sur seize *bits* (deux octets), mais dans la pratique il en occupe généralement trente deux (4 octets) ce qui permet de distinguer plusieurs milliards de valeurs. Il est censé avoir la taille convenable pour la plupart des usages courants (compter…) sur la machine considérée.

Ce type peut être précédé des mot-clefs `signed` ou `unsigned` pour indiquer s'il s'agit d'un entier relatif (`signed`) ou naturel (`unsigned`) ; par défaut un `int` est considéré `signed` c'est à dire qu'il peut représenter des valeurs positives ou négatives[10]. Bien que l'opportunité nous soit

9 L'optimisation par le compilateur peut faire qu'une variable ne soit pas du tout stockée dans la mémoire mais uniquement dans les registres du processeur (les matières S5-MIP et S6-MIP de l'ENIB sont consacrées à l'étude des microprocesseurs) ; cela ne change rien au fait que la variable a une valeur initiale indéterminée.

offerte, il est généralement déconseillé d'avoir recours au mot clef *unsigned* ; seulement dans quelques rares cas identifiés par des experts son usage s'impose.

Les constantes littérales de type *int* sont saisies dans le code source comme une séquence de caractères numériques exprimée en décimal (base dix). Toutefois, il est également possible de les saisir en hexadécimal (base seize) à l'aide du préfixe *0x*, ou encore en octal (base huit) à l'aide du préfixe *0*, lorsque cela semble adapté au contexte applicatif.

(http://en.cppreference.com/w/c/language/integer_constant)

```
int v1=1234, v2=-5678;      // entiers relatifs, constantes décimales
int v3=0xBadC0de, v4=-0644; // entiers relatifs, constante hexadécimale puis octale
unsigned int v5=16384;      // entier naturel, constante décimale
unsigned int v6=0xDeadBeef; // entier naturel, constante hexadécimale
```

Pour qu'un programme en langage C affiche la valeur d'un *int*, nous réutilisons la fonction *printf()* déjà présentée. Elle ne sert pas seulement à afficher du texte littéral ; la chaîne de caractères qui lui est transmise comme premier paramètre (la chaîne de format) peut contenir des séquences spéciales, préfixées par le symbole % (pourcent), afin d'insérer dans le texte affiché la valeur des paramètres supplémentaires dans l'ordre de leur énumération[11].

```
int v1=0xBadC0de, v2=-5678;
printf("v1 is %d and 10*v2 is %d\n",v1,10*v2); // v1 is 195936478 and 10*v2 is -56780
```

Ici, seule est utilisée la séquence *%d*, mais il existe, pour les entiers et bien d'autres types, énormément de nuances pour la mise en forme.

(http://en.cppreference.com/w/c/io/fprintf)

Le type *double* : les réels

Lorsque les entiers ne sont plus suffisants pour exprimer la précision de certains calculs arithmétiques, les types *float* et *double* permettent de manipuler des valeurs numériques à virgule flottante[12]. Bien que rien ne l'impose dans la norme du langage C, il s'agit très souvent dans le pratique du format IEEE754 en simple-précision (tenant sur quatre octets) et double-précision (tenant sur huit octets).

(https://en.wikipedia.org/wiki/IEEE_floating_point)

D'une manière générale il est recommandé de n'avoir recours qu'au type *double* ; seulement dans quelques rares cas identifiés par des experts l'usage du type *float* s'impose.

Les constantes littérales de type *double* sont saisies dans le code source comme une séquence de caractères numériques qui contient le symbole *.* (point, séparateur décimal) ou bien le caractère *e* pour signifier l'élévation à une puissance de dix.

(http://en.cppreference.com/w/c/language/floating_constant)

```
double v1=0.0, v2=12.34, v3=-56.78; // utilisation du séparateur décimal
double v4=-1e-6, v5=6.02e23;        // élévation à une puissance de dix
```

Pour qu'un programme en langage C affiche la valeur d'un *double*, nous réutilisons à nouveau la fonction *printf()* mais avec une nouvelle séquence spéciale dans la chaîne de format.

```
double v1=0.5e-4, v2=9.81;
printf("v1 is %g and v1*v2 is %g\n",v1,v1*v2); // v1 is 5e-05 and v1*v2 is 0.0004905
```

Ici, seule est utilisée la séquence *%g* mais il existe encore bien d'autres nuances pour la mise en forme.

(http://en.cppreference.com/w/c/io/fprintf)

---

10 Le mot-clef *signed* ne sert donc à rien devant *int*. Pour des raisons historiques, il est autorisé de s'affranchir du mot-clef *int* (qui devient implicite) si on utilise les mot-clefs *signed* ou *unsigned*.

11 Remarquez que la fonction *printf()* reçoit un nombre variable de paramètres ; malgré son apparente simplicité d'usage, cette fonction est une des plus compliquées de celles qui sont fournies en standard avec le langage C.

12 La distinction entre ces deux types, sera étudiée dans l'exercice pratique du chapitre 4.

Le type *char* : les caractères

Bien que la manipulation de données textuelles ne soit pas traitée dans l'immédiat mais dans un futur sujet d'exercice pratique (chapitre 5), nous pouvons déjà nous intéresser au type qui permet de représenter chacun des caractères d'un texte : le type *char*. Il ne s'agit finalement que d'un très petit entier qui occupe un unique *byte* (un octet dans la pratique). Comme tout type entier, il peut être précédé des mot-clefs *signed* ou *unsigned*[13]. Ce type s'utilise donc avec deux significations différentes : nous pouvons le considérer comme un très petit entier (relatif ou naturel) pour compter ou encore l'interpréter comme un caractère dans un texte. Dans tous les cas il ne s'agit que d'une valeur numérique tenant sur un octet. La conversion entre cette valeur numérique et un caractère n'est qu'affaire de convention. Il existe à ce propos une table de conversion unanimement reconnue : la table ***ASCII*** (vocabulaire).

(http://en.cppreference.com/w/c/language/ascii)

Dans ces conditions, un texte n'est rien d'autre qu'une séquence de telles valeurs numériques, tenant chacune sur un octet, et le périphérique qui les reçoit (imprimante, écran…) se contente de produire le dessin qui représente chacune d'elles : la réception du code *65* provoque l'affichage du caractère *A*, *122* le caractère *z*, *43* le caractère *+*… Parmi ces caractères, certains ne provoquent pas d'affichage immédiat mais modifient la disposition du texte. C'est le cas par exemple du code *10* qui fait changer de ligne ou du code *9* qui insère une tabulation. Vous remarquerez certainement que la table ASCII ne décrit que les codes de *0* à *127* alors que l'octet constituant un *char* peut désigner deux fois plus de valeurs distinctes. Les codes supplémentaires ne sont pas standards et peuvent parfois désigner des caractères accentués qui varient d'une culture à une autre[14]. La saisie de la valeur d'un *char* dans un code source ne nécessite heureusement pas la consultation de la table ASCII ; le langage C en a connaissance et produit le code numérique pour nous. Il nous suffit pour cela de saisir le caractère attendu entre une paire de symboles *'* (apostrophe).

(http://en.cppreference.com/w/c/language/character_constant)

```
char v1='E', v2='N', v3='I', v4='B'; // saisie des caractères littéraux
char v5=69, v6=78, v7=73, v8=66;     // saisie des codes ASCII (entiers)
```

La fonction *printf()* peut bien entendu afficher la valeur d'un *char* comme s'il s'agissait d'un entier, mais pour l'afficher sous la forme d'un caractère la séquence spéciale *%c* est nécessaire dans la chaîne de format.

(http://en.cppreference.com/w/c/io/fprintf)

```
char v1='E', v2='N', v3=73, v4=66;
printf("as text: %c %c %c %c\n",v1,v2,v3,v4);          // as text: E N I B
printf("as int: %d %d %d %d\n",v1,v2,v3,v4);           // as int: 69 78 73 66
printf("as text: %c %c %c %c\n",v1-4,v2-12,v3-6,v4+2); // as text: A B C D
```

Il est important de comprendre que la notation utilisant des apostrophes n'est qu'une façon alternative de saisir une valeur entière littérale dans le code source. Ainsi *'A'*, *65* ou encore *0x41* représentent tous exactement la même valeur, et *'A'+10* vaut *75* tout comme *'K'*. Les caractères non imprimables peuvent être saisis à l'aide du symbole ** (backslash) : par exemple *'\n'* pour changer de ligne ou *'\t'* pour insérer une tabulation.

(http://en.cppreference.com/w/c/language/escape)

Le type *bool* : les booléens

Historiquement, il n'y avait pas de type booléen (représentant uniquement vrai ou faux) en langage C. En effet, toute valeur numérique nulle (quel que soit sont type) est considérée comme fausse et par conséquent toute valeur numérique non nulle est considérée comme vraie (quelle que soit sa valeur précise : *5*, *8.4*, *-12*…). Dans ces conditions, un simple *int* ou *char* était suffisant pour représenter cette notion. Toutefois, afin de rendre le code plus

13 Le langage C ne précise pas si, privé d'un tel qualificatif, un *char* est *signed* ou *unsigned* (alors que *int* est *signed* par défaut).

14 L'étude des différents façons de représenter les jeux de codes de caractères dépasse de très loin l'objectif de cette prise en main du langage C.

explicite, le type *bool* et les constantes *true* et *false* ont été rendus disponibles dans la version C99 par l'inclusion du fichier d'en-tête *stdbool.h*.

Il n'existe cependant pas de moyen standard de les représenter lors de l'usage de la fonction *printf()* ; il suffit alors de les afficher comme des entiers qui vaudront *1* ou *0* selon le cas.

```
#include <stdbool.h>    // en dehors de toute fonction
...
bool v1=true, v2=false; // deux variables booléennes et les deux valeurs possibles
printf("v1 is %d and v2 is %d\n",v1,v2); // v1 is 1 and v2 is 0
```

Le mot-clef *typedef* : inventer de nouveaux types

Il existe une facilité du langage qui permet d'inventer un nouveau nom pour désigner un type. Cela permet par exemple d'alléger l'écriture du code source (gain en confort) ou encore de rendre plus explicites à la lecture les intentions liées à l'utilisation de certaines données.

```
typedef unsigned char Byte; // Byte est le nom d'un nouveau type identique à unsigned char
typedef double Mass;        // Mass est le nom d'un nouveau type identique à double
...
Byte v1=7, v2=130; // ces deux variables ne sont rien d'autre que des unsigned char
Mass v3=26.4;      // celle ci n'est rien d'autre qu'un double
```

L'utilisation du mot clef *typedef* précède simplement une formulation semblable à une déclaration de variable ; cependant le nom choisi ne désigne pas une variable mais un nom alternatif pour ce type. Ces constructions sont généralement placées à l'extérieur de toute fonction ce qui fait qu'elles sont visibles pour toute la suite du code[15]. Dans l'exemple, le nom *Byte* a été choisi pour autoriser une économie d'écriture alors que *Mass* insiste sur le fait que les variables ayant ce type s'interprètent comme des masses (et non des longueur ou des vitesses).

Les quelques types usuels présentés ici (*int*, *double*, *char*, *bool*) sont suffisants pour aborder les rudiments du langage C. Les propriétés de l'éventail des types arithmétiques du langage C sont détaillées dans ce document :

(http://en.cppreference.com/w/c/language/arithmetic_types)

Un sujet d'exercice pratique (chapitre 4) sera consacré à l'expérimentation autour des propriétés des nombreuses variantes de ces types.

1.3.3. Conversion de type

Dans tous nos exemples précédents, nous n'utilisions à chaque fois que des variables et des valeurs de type cohérent. Il arrive néanmoins qu'il soit nécessaire de faire intervenir des types différents dans un même calcul. Deux cas de figure se présentent :

- si le compilateur sait que la transmission d'une valeur d'un type à un autre peut se faire sans perte d'information, alors la conversion aura lieu implicitement,

  (http://en.cppreference.com/w/c/language/conversion)
- si au contraire le compilateur soupçonne un risque de perte d'information, alors il pourra générer un message d'avertissement lors de la compilation[16].

Dans ce second cas, nous devons signifier qu'une telle conversion est intentionnelle. Cela ne concerne pas uniquement le compilateur, qui n'affichera plus ce message d'avertissement dorénavant, mais s'adresse surtout aux développeurs qui, lors de leur relecture, sauront que l'éventuelle perte d'information a été étudiée et maîtrisée par l'auteur de ces lignes de code.

15 Bien que ce soit plus rare, il est possible d'en faire usage dans un bloc de code (le corps d'une fonction par exemple) ; ce nouveau nom de type n'est alors visible que jusqu'à la fin du bloc en question.

16 La rigueur du diagnostic des différents compilateurs est variable mais il est recommandé de leur transmettre des options réclamant une grande sévérité afin de détecter au plus tôt les maladresses potentielles.

Cette conversion explicite de type est désignée par le terme ***coercition*** (vocabulaire) ou plus communément ***cast*** (*type-cast*, vocabulaire). Elle s'exprime en faisant précéder une expression par le nouveau type choisi entre les symboles `( )` (parenthèses).

```
double v1=12.34;
int v2=v1, v3=(int)v1;                          // avertissement pour v2, pas pour v3
unsigned int v4=v2, v5=(unsigned int)(v2+v3); // avertissement pour v4, pas pour v5
double v6=v5;                                   // aucun avertissement
printf("v1=%g v6=%g \n",v1,v6);                    // v1=12.34 v6=24
printf("v2=%d v3=%d v4=%d v5=%d \n",v2,v3,v4,v5); // v2=12 v3=12 v4=12 v5=24
```

Remarquez qu'aucun avertissement n'est produit lorsque la variable *v6* est initialisée depuis *v5* car aucune information n'est perdue à cette occasion ; bien que correct, un *cast* aurait été inutile dans ce cas. Il est important de comprendre qu'une conversion ne modifie pas la donnée d'origine mais en produit une copie dont le type est changé (la variable d'origine garde bien évidemment sa valeur et son type).

(http://en.cppreference.com/w/c/language/cast)

Les conversions, qu'elles soient implicites ou explicites, ne se traduisent pas forcément par des instructions supplémentaires dans le code exécutable généré par le compilateur. Dans le cas d'une conversion entre un nombre à virgule flottante et un entier il y a bien une instruction de calcul supplémentaire à exécuter car ces deux représentations sont très différentes du point de vue de leurs motifs de *bits*. En revanche, entre un entier naturel et relatif de même taille les représentations sont identiques et les opérations de calcul sont pour la plupart similaires ; le changement de type ne fait qu'influencer la manière d'interpréter certaines valeurs (négatives ou très grandes positives) mais n'ajoute pas d'instruction supplémentaire.

1.3.4. Portée et classe de stockage des variables

Les variables que nous avons présentées ont toutes une ***portée locale*** (vocabulaire) à la fonction dans laquelle elles sont déclarées. Elles ne sont visibles que dans la fonction à laquelle elles appartiennent ; des variables, ou des paramètres, de même nom dans deux fonctions différentes sont complètement distincts. Ces variables locales suffisent généralement à exprimer et à traiter l'ensemble des problèmes courants et ce sont elles qu'il faut utiliser en priorité.

Il existe néanmoins d'autres variables dont la ***portée*** est ***globale*** (vocabulaire). Celles-ci sont déclarées en dehors de toute fonction et sont de ce fait accessibles depuis le code de n'importe quelle fonction. Toutefois, si dans une fonction une variable locale a le même nom qu'une variable globale, seulement la variable locale sera visible dans cette fonction (elle masque la variable globale).

Nous avons déjà mentionné que les variables locales que nous avons utilisées jusqu'alors avaient une ***classe de stockage automatique*** (vocabulaire) : elles apparaissent avec une valeur indéterminée lorsque l'exécution atteint le bloc de code (paire d'accolades) dans lequel elle sont déclarées et disparaissent en en sortant. Puisqu'au contraire les variables globales existent en dehors des fonctions, elles sont créées une fois pour toutes au début du programme et restent accessibles jusqu'à sa terminaison : elles ont une ***classe de stockage statique*** (vocabulaire). Cette propriété laisse envisager la possibilité de partager des données entre plusieurs fonctions en les plaçant dans des variables globales ; ***il s'agit d'une très mauvaise pratique qui est fortement déconseillée !*** Ceci conduit à un enchevêtrement des actions des fonctions qui rend le code impossible à maintenir. Seulement dans quelques rares cas identifiés par des experts cette solution peut être justifiée. Le problème de la distinction entre la déclaration et la définition d'une variable globale se pose dans les mêmes termes qu'en ce qui concerne les fonctions ; toutefois, puisque l'usage des variables globales est fortement déconseillé, nous n'aborderons pas ce point ici.

Il serait tentant d'associer systématiquement la notion de variable globale à la classe de stockage statique et la notion de variable locale à la classe de stockage automatique[17], mais cette vision est erronée. Une variable locale peut avoir sa déclaration précédée du mot-clef `static`. Dans ce cas, elle reçoit sa valeur initiale (ou `0` par défaut) lors du démarrage du programme et conserve sa valeur (qui peut être modifiée) d'une invocation de la fonction à une autre. Elle a donc bien une classe de stockage statique puisqu'elle existe continûment pendant toute la durée du programme, mais elle n'en est pas moins locale car elle ne peut être accédée que depuis la fonction à laquelle elle appartient.

```
#include <stdio.h>

void
testStatic(int i)
{
static int s=100;   // variable initialisée au démarrage du programme
s=s+i;              // variable modifiée à chaque nouvelle invocation
printf("s=%d\n",s);
}

int
main(void)
{
testStatic(1);  // s=101
testStatic(10); // s=111
return 0;
}
```

Bien que leur usage ne soit pas aussi mauvais que celui des variables globales en ce qui concerne la maintenance du code, l'utilisation des variables locales statiques est tout de même plutôt déconseillé[18].

Ces documents apportent des détails sur la portée et les classes de stockage :

(http://en.cppreference.com/w/c/language/scope)

(http://en.cppreference.com/w/c/language/storage_duration)

1.4. Les règles d'écriture du langage

Nous abordons maintenant plus en détail la forme des constructions qui constituent un programme en langage C.

1.4.1. Le préprocesseur

Avant de jouer son rôle, le compilateur de langage C s'appuie sur un autre langage bien plus rudimentaire : le ***préprocesseur*** (vocabulaire). Son propos est de modifier à la volée le code source que le compilateur analysera effectivement. Lorsque nous invoquons le compilateur, celui-ci invoque pour nous le préprocesseur et analyse enfin le code en langage C résultant de ces transformations préalables. De tels interventions sont qualifiées de ***transformations lexicales*** (vocabulaire) dans le sens où elles se contentent de remplacer des portions de texte par d'autres en ignorant la signification de ce texte pour le langage C.

Dans un code source, les lignes qui concernent le préprocesseur sont celles qui commencent par le symbole `#` (dièse) ; il s'agit de ***directives*** (vocabulaire). Puisque le préprocesseur intervient avant l'analyse des détails du langage C, de telles directives peuvent intervenir n'importe où dans le code (à l'extérieur ou à l'intérieur des fonctions).

17 Cette confusion est très répandue parmi les programmeurs débutants.

18 La principale limitation de la classe de stockage statique concerne l'impossibilité d'en faire usage dans du code dont l'exécution parallélisée vise à exploiter les multiples unités de calcul d'une machine informatique.

La directive `#include` : l'inclusion de fichier

Nous avons déjà rencontré l'usage de la directive `#include` dans l'exemple du paragraphe 1.2.4. Son rôle consiste à lire le contenu du fichier indiqué et à l'insérer dans le code source à l'endroit de cette directive comme si nous avions eu recours à copier/coller.

```
#include <stdio.h>    // inclusion d'un fichier standard, pré-installé dans le système
#include "myheader.h" // inclusion d'un fichier propre à notre projet
```

Le procédé est récursif, c'est à dire qu'un fichier inclus peut lui-même en inclure d'autres. L'usage des fichiers d'en-tête concernés par ces inclusions et des symboles `< >` (chevrons) ou `"` (guillemets) est discuté dans un sujet d'exercice pratique consacré à la construction des projets (chapitre 2).

(http://en.cppreference.com/w/c/preprocessor/include)

La directive `#define` : la définition de macros

Un autre usage très courant du préprocesseur consiste à utiliser la directive `#define` pour définir des ***macro-instructions*** (***macros***, vocabulaire).

```
#define COUNT 1000      // le mot COUNT sera substitué par la constante 1000
#define WEIRD_SUM a + b // le mot WEIRD_SUM sera substitué par l'expression a + b

int a=COUNT, b=2*COUNT; // a reçoit 1000 et b 2000
printf("WEIRD_SUM is %d\n",WEIRD_SUM); // WEIRD_SUM is 3000
```

Dans la suite du code source, le nom d'une *macro*[19] est alors substitué par tout ce qui constitue la fin de la ligne (à l'exception des commentaires) ; cette substitution n'a cependant pas lieu dans le texte littéral. Comme illustré ici par la *macro* `COUNT`, il s'agit d'un moyen confortable pour définir une constante en un endroit unique du code source plutôt que d'avoir à en répliquer explicitement la valeur à de multiples reprises ; cela facilite notamment sa modification. Veuillez toutefois remarquer que la substitution peut concerner un texte quelconque qui peut même s'avérer incorrect en langage C ; c'est le cas de l'exemple absurde `WEIRD_SUM` qui n'a de sens que si les variables `a` et `b` existent à l'endroit de l'utilisation de cette *macro*.

Les *macros* autorisent des substitutions plus subtiles qui prennent en compte des paramètres.

```
#define BAD_SUM(x,y)        x+y       // aucune protection --> interaction possible
#define BETTER_SUM(x,y)     ((x)+(y)) // protection de toute l'expression et de ses paramètres
#define BAD_PRODUCT(x,y)    x*y       // aucune protection --> interaction possible
#define BETTER_PRODUCT(x,y) ((x)*(y)) // protection de toute l'expression et de ses paramètres

int v1=10, v2=100;
printf("2*BAD_SUM(%d,%d) is %d\n",        // 2*BAD_SUM(10,100) is 120
       v1,v2,   2*BAD_SUM(v1,v2));        // 2*v1+v2
printf("2*BETTER_SUM(%d,%d) is %d\n",     // 2*BETTER_SUM(10,100) is 220
       v1,v2,   2*BETTER_SUM(v1,v2));     // 2*((v1)+(v2))
printf("BAD_PRODUCT(%d,%d) is %d\n",      // BAD_PRODUCT(12,100) is 210
       v1+2,v2, BAD_PRODUCT(v1+2,v2));    // v1+2*v2
printf("BETTER_PRODUCT(%d,%d) is %d\n",   // BETTER_PRODUCT(12,100) is 1200
       v1+2,v2, BETTER_PRODUCT(v1+2,v2)); // ((v1+2)*(v2))
```

Comme illustré avec les *macros* `BAD_SUM` et `BAD_PRODUCT` de cet exemple, la substitution lexicale peut faire apparaître des interactions (inattendues à la lecture du code) entre le texte substitué, ses paramètres et ce qui suit ou précède. Une solution pour limiter ces risques consiste à s'imposer de protéger systématiquement l'expression et les paramètres dans le contenu de la *macro*. À cause des risques de confusion illustrés ici, il est généralement

19 Bien que rien ne l'impose, il est extrêmement courant de n'utiliser que des majuscules dans un nom de *macro* ; cela attire l'attention du lecteur sur le fait qu'il s'agit d'une *macro* et non d'une simple variable.

déconseillé d'avoir recours à de telles *macros* au contenu élaboré ; elles servent généralement à introduire de manière optionnelle des fonctionnalités qui aident à la mise au point (trace, débogage) mais ne font généralement pas partie du code utile du programme.

(http://en.cppreference.com/w/c/preprocessor/replace)

La directive `#if` : la compilation conditionnelle

La définition de *macros* offre la possibilité de choisir des portions de code qui seront compilées ou non selon les circonstances (plateforme d'exécution visée, compilateur disponible, phase de mise au point...). Il s'agit d'effectuer de la ***compilation conditionnelle*** (vocabulaire).

```
#if defined _WIN32 // la macro _WIN32 n'existe que si nous compilons pour Windows
... ces lignes contiennent du code spécifique à Windows
... elles ne seront compilées que pour cette plateforme
#else
... celles-ci contiennent du code pour les autres systèmes
#endif
```

Les *macros* sur lesquelles porte la directive `#if` peuvent être définies dans le code source ou par le compilateur[20]. Les expressions portent non seulement sur l'existence de ces *macros* mais également sur leur contenu et peuvent faire intervenir des constantes et des opérations.

(http://en.cppreference.com/w/c/preprocessor/conditional)

Au delà de ces quelques fonctionnalités d'usage très courant, le préprocesseur en propose quelques autres :

(http://en.cppreference.com/w/c/preprocessor)

1.4.2. La mise en forme

Contrairement au langage Python, la mise en forme d'un code source en langage C est très libre. Bien que ce soit une très mauvaise idée pour la maintenance du code, nous pouvons avoir le loisir de tout écrire sur une seule ligne (à l'exception des directives du préprocesseur qui occupent nécessairement une ligne) ! Il est au contraire recommandé d'aérer le code en sautant des lignes pour séparer les portions distinctes et en indentant comme Python l'impose. Lorsqu'une forme de présentation a été choisie, il est important de s'y tenir pour faciliter la lecture du code.

Les commentaires sont filtrés avant l'action du préprocesseur. Ils ne servent qu'à informer le lecteur sur la finalité du code ou ses subtilités de fonctionnement. Il faut les utiliser autant que possible pour expliciter ce qui n'est pas évident à la lecture en veillant à ne pas commenter ce qui est trivial pour un programmeur moyen.

Ils sont de deux formes. Nous avons déjà utilisé dans les exemples le symbole `//` (double-*slash*) qui neutralise le texte jusqu'à la fin de la ligne. La seconde forme permet d'exprimer des commentaires qui occupent plusieurs lignes sans avoir à répéter un symbole sur chacune d'elles. Il suffit de saisir le symbole `/*` (*slash*-étoile), le texte du commentaire sur une ou plusieurs lignes et enfin le symbole `*/` (étoile-*slash*).

(http://en.cppreference.com/w/c/comment)

20 Sous UNIX, une commande telle que `echo "" | gcc -dM -E - | sort` nous permet de prendre connaissance des *macros* qui sont automatiquement définies par le compilateur invoqué.

```
// ce commentaire doit nécessairement tenir sur une seule ligne
/* cette autre forme de commentaire fonctionne aussi sur une seule ligne */
/* mais elle a l'intérêt de s'étendre facilement sur plusieurs lignes.
Attention toutefois : en saisissant à nouveau le symbole /* nous pouvons penser à une
imbrication de commentaires mais il n'en est rien. Ce long commentaire se termine bien ici. */
#if 0  // cette directive émule un commentaire sur plusieurs lignes
  printf("I don't want this code to be compiled!\n");
# if 0 // et elle offre l'avantage de permettre l'imbrication.
    printf("Not even this one...\n");
# endif
#endif
```

Remarquez sur cet exemple l'usage de la directive `#if` pour émuler des commentaires qui peuvent être imbriqués. Cette solution offre de plus l'avantage d'activer ou désactiver à la demande des portions de code, lors de la mise au point du programme par exemple, en basculant un unique caractère (`#if 0` ou `#if 1`).

1.4.3. Les identificateurs

Au fil des exemples rencontrés jusqu'alors, il nous a été nécessaire de choisir des noms pour identifier des fonctions, des variables, des nouveaux types (`typedef`), et des *macros*. Malgré une très grande liberté dans le choix de ces ***identificateurs*** (vocabulaire), il y a néanmoins quelques règles à respecter.

Un identificateur est constitué d'une séquence non vide de caractères alphabétiques (de `a` à `z` et de `A` à `Z`, en distinguant les majuscules des minuscules), numériques (de `0` à `9`) ou `_` (souligné). Le premier caractère ne peut pas être numérique. De plus, certaines combinaisons incluant notamment plusieurs symboles `_` (souligné) à la suite sont réservées pour l'usage interne de certains compilateurs.

(http://en.cppreference.com/w/c/language/identifier)

Il ne peut, bien entendu, pas non plus s'agir d'un mot-clef du langage :

(http://en.cppreference.com/w/c/keyword)

Il est important de savoir que les identificateurs ne sont absolument pas considérés dans des espaces de noms distincts selon la nature de ce qu'ils désignent. Si dans un unique contexte (un bloc de code par exemple) plusieurs déclarations font apparaître un même identificateur, il y alors une collision de noms que le compilateur signalera par une erreur.

```
typedef double value; // le nom value désigne ici un type
value x=2.3;          // x est une variable de type value (ie double)
int value=8;          // erreur de compilation ! (value existe déjà)
int printf=12;        // la variable locale printf masque la fonction de même nom
printf("really?\n");  // erreur de compilation ! (printf n'est pas une fonction dans ce contexte)
```

Il parait assez évident que deux variables du même contexte ne puissent avoir le même nom, mais le problème se pose aussi entre une variable et un type par exemple. Lorsque la collision de nom concerne des contextes imbriqués, le contexte local d'une fonction et le contexte global par exemple (voir en 1.3.4), alors les identificateurs du contexte imbriqué masquent ceux du contexte externe. Dans cet exemple la variable locale `printf` masque la fonction de même nom qui existe dans le contexte global et devient de ce fait inaccessible.

1.4.4. Les blocs, les instructions et les expressions

Nous avons déjà vu que le corps de la définition d'une fonction pouvait renfermer des déclarations de variables et des instructions pour les modifier ou les afficher par exemple.

De tels ***blocs*** (vocabulaire), matérialisés par les symboles `{ }` (accolades), peuvent apparaître partout ou une instruction est attendue ; il est donc possible de les imbriquer, ce qui sera particulièrement utile pour les instructions de contrôle (voir en 1.4.5)[21].

21 Un bloc vide est valide, tout comme un simple point-virgule : c'est une instruction sans effet.

```
{
int v1=6, v2=8;
printf("v1 is %d and v2 is %d\n",v1,v2);                                  // v1 is 6 and v2 is 8
  { // ouverture d'un nouveau bloc (inutile ici, juste pour l'exemple)
  int v1=12; // cette nouvelle variable v1 masque celle du bloc englobant
  printf("v1 is %d and v2 is %d\n",v1,v2);                                // v1 is 12 and v2 is 8
  v1=v1+2; // modification de la variable locale à ce bloc
  v2=v2-5; // modification de la variable locale au bloc englobant
  }
printf("v1 is %d and v2 is %d\n",v1,v2);                                  // v1 is 6 and v2 is 3
}
```

Les variables déclarées dans un bloc sont locales à ce bloc et sont susceptibles de masquer celles d'un bloc englobant. Il est recommandé de déclarer les variables au plus près des instructions qui en font l'usage[22].

La forme des déclarations de variables a été décrite au paragraphe 1.3.1 et nous avons déjà rencontré quelques instructions simples dans les exemples (appels de fonction, modification de variables). Remarquez que toutes ces formulations se terminent par un symbole `;` (point-virgule) ; c'est obligatoire pour les distinguer (puisque l'indentation ne joue pas ce rôle, contrairement à Python).

(http://en.cppreference.com/w/c/language/statements)

À part en ce qui concerne les instructions de contrôle (voir en 1.4.5), le langage C ne fait pas vraiment de distinction entre une instruction et une expression. Une ***expression*** (vocabulaire) est une évaluation qui produit un résultat ; une ***instruction*** (vocabulaire) n'est alors rien d'autre qu'une expression dont nous ignorons la valeur résultante. Une instruction doit normalement produire un ***effet de bord*** (*side-effect*, vocabulaire), c'est à dire que si elle ne fournit pas directement de résultat exploitable, elle doit indirectement produire un effet qui soit observable par ailleurs (modification de données en mémoire, affichage...)[23].

```
printf("as an instruction\n");          // le résultat de l'appel à printf() est ignoré
int v1=10+printf("as an expression\n"); // ici il participe à une expression arithmétique
int v2=100;
v1=v1*2+8;           // cette instruction affecte le résultat d'une expression à une variable
v2=v2*3+(v1=v1*2+8); // une affectation est aussi une expression qui vaut la valeur affectée
printf("5=%d v1=%d v1+v2/2=%d\n", 5, v1, v1+v2/2);
```

En plus de fournir un résultat[24], l'effet de bord de la fonction `printf()` est la production d'un affichage. L'effet de bord d'une affectation est la modification de son membre gauche (***lvalue***, vocabulaire) ; il s'agit également d'une expression valant la nouvelle valeur du membre gauche. Bien entendu, pour qu'un appel de fonction soit utilisé comme une expression elle doit fournir un résultat : le type de sa valeur de retour ne doit pas être `void`. Les paramètres effectifs d'un appel de fonction sont eux-mêmes des expressions (constantes, variables, expressions arithmétiques...).

La forme des expressions est très variée. Il existe en particulier les opérateurs arithmétiques usuels : `+` (addition), `-` (soustraction ou opposé), `*` (multiplication), `/` (division), `%` (modulo, reste de la division entière). Dans la pratique, ils sont très couramment combinés avec l'opérateur `=` (affectation).

22 Dans les versions historiques du langage C il était nécessaire de les déclarer en tout début de bloc mais ce n'est désormais plus le cas.

23 Si les instructions, pour être utiles, doivent logiquement provoquer des effets de bord, les expressions peuvent elles-aussi en provoquer en plus de la valeur qu'elles produisent.

24 La fonction `printf()` renvoie le nombre de caractères qu'elle a produits en affichant le message.

```
int v1=10, v2=20;
v1+=2;    // équivalent à v1=v1+2;
v2*=v1+8; // équivalent à v2=v2*(v1+8);
```

Cette notation qui consiste à faire précéder d'une opération le symbole d'affectation est assez emblématique du langage C[25]. Elle est disponible pour tous les opérateurs arithmétiques à deux arguments.

Une autre forme d'opération très remarquable en langage C concerne l'incrémentation et la décrémentation des entiers[26].

```
int value, v1=10, v2=20, v3=30, v4=40;
value=++v1;                          // pré-incrémentation, équivalent à value=(v1+=1);
printf("v1=%d value=%d\n",v1,value); // v1=11 value=11
value=--v2;                          // pré-décrémentation, équivalent à value=(v2-=1);
printf("v2=%d value=%d\n",v2,value); // v2=19 value=19
value=v3++;                          // post-incrémentation, équivalent à value=v3; puis v3+=1;
printf("v3=%d value=%d\n",v3,value); // v3=31 value=30
value=v4--;                          // post-décrémentation, équivalent à value=v4; puis v4-=1;
printf("v4=%d value=%d\n",v4,value); // v4=39 value=40
```

Qu'on utilise une pré-opération ou une post-opération, la variable concernée est bien modifiée d'une unité. Il s'agit de l'effet de bord principalement attendu si on l'invoque comme une simple instruction. La différence ne se situe qu'au niveau de la valeur obtenue par l'évaluation d'une telle expression : l'effet d'une pré-opération a lieu avant d'extraire la valeur de la variable modifiée, alors qu'une post-opération ne modifie la variable qu'après l'extraction de sa valeur (cette différence subtile prend tout son intérêt dans les boucles de calcul).

Les expressions peuvent contenir les opérations de comparaison usuelles : `<` (inférieur à), `<=` (inférieur ou égal à), `>` (supérieur à), `>=` (supérieur ou égal à), `==` (égal à), `!=` (différent de). Elles donnent des résultats booléens qui peuvent être combinés avec des opérateurs logiques : `&&` (et), `||` (ou), `!` (non).

```
int v1=10, v2=20, v3=30, v4=40;
bool b1=v1>v2;
bool b2=(v1+v2==v3)&&(v3<=v4);
bool b3=!b2||(!b1&&(v4-v3!=v2));
printf("b1=%d b2=%d b3=%d\n",b1,b2,b3); // b1=0 b2=1 b3=1
```

Ces expressions logiques seront particulièrement utiles pour les instructions de contrôle (voir en 1.4.5).

Il est important de savoir que les opérateurs logiques `&&` et `||` sont évalués de manière paresseuse : cela signifie que si le membre de gauche permet déjà de déterminer le résultat de l'opération logique, alors le membre de droite ne sera pas évalué. C'est le cas si le membre de gauche est faux pour l'opérateur `&&` (le résultat est alors forcément faux) ou s'il est vrai pour l'opérateur `||` (le résultat est alors forcément vrai).

```
int v1=10, v2=20;
bool b1=(v1>v2)&&(printf("not displayed\n")>0);
bool b2=(v1>v2)||(printf("displayed\n")>0);      // displayed
bool b3=(v1<v2)&&(printf("displayed\n")>0);      // displayed
bool b4=(v1<v2)||(printf("not displayed\n")>0);
printf("b1=%d b2=%d b3=%d b4=%d\n",b1,b2,b3,b4); // b1=0 b2=1 b3=1 b4=1
```

25 Ceci est lié au fait qu'un processeur ne produit généralement pas une nouvelle valeur à partir de deux autres ; il se contente d'en modifier une selon la valeur d'une autre. Cette notation en langage C correspond directement à l'utilisation d'une telle instruction dans un processeur.

26 À nouveau, l'ajout ou le retrait d'une unité sur un entier constitue une opération élémentaire du processeur.

Il s'agit d'une optimisation qui consiste à ne pas évaluer inutilement ce qui ne changera rien au résultat de l'opération logique. Néanmoins, cela a une conséquence fondamentale sur les effets de bord de l'expression du membre droit : ils n'ont pas lieu de manière systématique mais sont déclenchés ou non selon la valeur logique du membre gauche.

Pour ajouter un peu plus de confusion à cela, il faut noter que l'ordre d'exécution des effets de bord n'est pas toujours défini !

(http://en.cppreference.com/w/c/language/eval_order)

Un aspect surprenant du langage C tient dans le fait qu'une même expression peut combiner des opérations arithmétiques, des comparaisons et des opérations logiques[27] dans n'importe quel ordre : l'ensemble forme une expression arithmétique et logique.

(http://en.cppreference.com/w/c/language/expressions)

Si un membre d'une opération arithmétique a un résultat logique, il sera considéré comme valant `0` ou `1` pour la suite du calcul arithmétique. De la même façon, si un membre d'une opération logique a un résultat arithmétique, il sera considéré comme faux si cette valeur est nulle et vrai dans tous les autres cas (quelle que soit la valeur arithmétique précise).

```
int v1=10, v2=20;
bool b1=v1&&!v2;                        // faux puisque v1!=0 est vrai et v2==0 est faux
bool b2=v1<v2;                          // vrai
int v3=b1+80*b2+500*(2*v1==v2);         // b1 vaut 0, b2 vaut 1, 2*v1==v2 est vrai et vaut 1
printf("b1=%d b2=%d v3=%d\n",b1,b2,v3); // b1=0 b2=1 v3=580
```

S'il est assez courant d'utiliser un résultat arithmétique dans une opération logique (on évite de s'encombrer de `==0` ou `!=0`, comme dans le calcul de `b1` ici), il est en revanche rarement utile de combiner des résultats logiques dans une opération arithmétique (comme le calcul de `v3` ici).

Toutes les opérations qui interviennent dans une expression sont soumises à des règles de priorité et d'associativité. Au delà des habitudes que chacun connaît à propos de l'arithmétique et de la logique, les multiples combinaisons avec des opérateurs de toutes sortes peuvent conduire à des expression difficiles à interpréter pour un programmeur qui lirait le code. Il est alors recommandé d'expliciter les associativités et les priorités des calculs par l'usage de parenthèses.

(http://en.cppreference.com/w/c/language/operator_precedence)

1.4.5. Les instructions de contrôle

Les instructions de contrôle sont spéciales dans le sens où elles ne peuvent pas prendre place dans une expression. Elles servent principalement à produire des branchements (des sauts dans l'exécution depuis une portion de code vers une autre) et des boucles.

L'instruction `if` : l'alternative simple

Un moyen très courant de provoquer un branchement consiste à utiliser une ***alternative simple*** (vocabulaire)[28]. L'évaluation de la valeur logique d'une expression conditionne l'exécution d'un unique bloc de code.

```
if(v1<v2)                               // if(v1<v2) printf("v1 is less than v2\n");
  {
  printf("v1 is less than v2\n");
  }
```

Notez que les parenthèses englobant la ***condition*** (vocabulaire) après le mot-clef `if` sont obligatoires. En revanche, les accolades marquent simplement un bloc d'instructions ; si une seule instruction est nécessaire, celle-ci n'a pas besoin de figurer dans un bloc. Toutefois, il est recommandé de faire usage d'un bloc si cela facilite la lecture du code.

27 Et d'autres qui seront étudiées au chapitre 9.

28 Cette notion a été étudiée en détail dans la matière S1-ALR de l'ENIB.

L'alternative simple permet également de choisir l'exécution d'un bloc de code parmi deux en fonction de la condition.

```
if(v1<v2)
  {
  printf("v1 is less than v2\n");
  }
else
  {
  printf("v1 is greater than or equals to v2\n");
  }
```

(http://en.cppreference.com/w/c/language/if)

Puisqu'il s'agit d'instructions, de telles alternatives simples peuvent être imbriqués pour évaluer plusieurs conditions exclusives à la suite :

```
if(v1<v2)                                    // if(v1<v2)
  { printf("v1 is less than v2\n"); }        //   printf("v1 is less than v2\n");
else if(v1>v2)                               // else
  { printf("v1 is greater than v2\n"); }     //   if(v1>v2)
else                                         //     printf("v1 is greater than v2\n");
  { printf("v1 equals to v2\n"); }           //   else
                                             //     printf("v1 equals to v2\n");
```

Il ne s'agit pas d'une construction particulière du langage : dans ce cas la deuxième instruction `if` est simplement l'unique instruction associée à la première instruction `else`[29].

Cet exemple illustre le fait que, contrairement au langage Python, l'indentation n'a aucune influence sur la signification du code. En toute rigueur, et comme illustré en commentaire, le dernier `if` et le dernier `else` devraient être alignés et indentés sous le premier `else`. Cependant, le style d'indentation initialement choisi exprime clairement l'intention première du programmeur : envisager les trois cas distincts pour comparer les deux variables.

Bien que l'instruction `if` ne puisse pas figurer dans une expression (elle n'a pas de valeur, elle ne peut que provoquer des effets de bord), il existe néanmoins un opérateur qui lui ressemble : l'***opérateur ternaire*** (ou ***opérateur conditionnel***, vocabulaire) représenté par les symboles `?` (point d'interrogation) et `:` (deux-points). Il est qualifié de ternaire car il dispose de trois membres : le premier (avant `?`) est interprété comme une condition, le deuxième (entre `?` et `:`) représente l'expression qui sera évaluée si la condition est vérifiée et le troisième (après `:`) l'expression qui sera évaluée sinon. Une seule des deux dernières expressions est évaluée ; l'autre ne produira donc pas ses éventuels effets de bord. Bien entendu, ces deux dernières expressions doivent être de même type car l'évaluation d'un tel opérateur ternaire fournit un résultat d'un unique type.

```
int v1=10, v2=20;                                  // int tmp;
int v3=v1<v2 ? v1*500 : v2+500;                    // if(v1>v2) { tmp=v1; }
int v4=2*(v1>v2 ? v1 : v2)+5;                      // else      { tmp=v2; }
printf("v3=%d v4=%d\n",v3,v4);  // v3=5000 v4=45   // int v4=2*tmp+5;
```

La réécriture sous forme de commentaires de l'initialisation de la variable `v4` de cet exemple illustre l'intérêt de cet opérateur. Il permet d'exprimer de façon plus concise, et finalement plus lisible, des formules de calcul dont des portions dépendent d'expressions logiques. Il est toutefois fort probable que le code généré par le compilateur soit très similaire dans les deux cas (opérateur ternaire ou alternative simple) ; cette facilité sert essentiellement au confort de rédaction et de lecture des programmeurs.

(http://en.cppreference.com/w/c/language/operator_other)

29 Il n'existe pas en langage C d'instruction `elif` telle que proposée par Python.

L'instruction `switch-case` : l'alternative multiple

En complément de l'alternative simple, le langage C propose l'***alternative multiple*** (vocabulaire). Il s'agit d'évaluer une expression une seule fois et de provoquer un branchement vers diverses portions de code selon la valeur obtenue. Chaque portion de code pouvant être visée par un tel branchement est désignée par une étiquette correspondant à une valeur constante littérale (c'est à dire connue du compilateur à l'analyse du code) de type entier[30].

```
int data=obtainAnyData();
switch(data*2+3)
  {
  case 3:
    { printf("this case was expected\n"); break; }
  case 5:
    { printf("so was this one\n");        break; }
  case 7:
    { printf("and this one too\n");       break; }
  default:
    { printf("unexpected case!\n");       break; }
  }
```

Sur cet exemple, l'évaluation de l'expression qui accompagne l'instruction `switch` provoquera l'affichage d'un des trois premiers messages selon que la valeur obtenue vaut `3`, `5` ou `7` ; pour toute autre valeur c'est le dernier message qui sera affiché.

Tout comme pour l'instruction `if`, les parenthèses englobant l'expression après le mot-clef `switch` sont obligatoires. Le bloc qui suit est ponctué d'étiquettes constituées du mot-clef `case`, d'une constante entière et du symbole `:` (deux-points). Ces étiquettes sont suivies de zéro, une ou plusieurs instructions qui seront exécutées si l'expression évaluée correspond à l'étiquette. L'usage d'un bloc à cet endroit n'est pas nécessaire, même s'il y a plusieurs instructions ; toutefois, il est recommandé d'en faire usage si cela facilite la lecture du code[31]. Remarquez que ces instructions sont terminées par l'instruction de contrôle `break` ; elle n'est pas strictement obligatoire mais son omission provoque un résultat tout à fait inattendu qui est une source d'erreur extrêmement classique. En effet, sans cette instruction l'exécution du code se poursuit avec les instructions de l'étiquette suivante ! Il n'y a en pratique que quelques cas exceptionnellement rares pour lesquels ce comportement est satisfaisant. Il faut donc systématiquement terminer le code associé à chaque étiquette par l'instruction `break` qui provoque la sortie immédiate du bloc qui constitue le corps de l'instruction `switch`. L'étiquette représentée par le mot-clef `default` n'est associée à aucune constante particulière et permet de fournir les instructions à exécuter si aucune autre étiquette ne correspond à la valeur évaluée ; elle est optionnelle, et en cas d'absence le corps du `switch` n'exécutera aucune instruction.

D'une manière générale, ***l'instruction `switch-case` est considérée comme dangereuse pour les programmeurs débutants en langage C*** ; elle est source d'erreurs difficiles à déceler. Il est déconseillé de s'en servir puisqu'il n'y a finalement que très peu de cas pratiques pour lesquelles son usage apporte un avantage par rapport à une imbrication d'alternatives simples telles que vues un peu plus haut.

L'instruction `while` : la boucle à pré-condition

Pour provoquer la répétition d'une portion de code, il est commun d'avoir recours à une boucle itérative. Il en existe quelques variantes en langage C ; celle qui est identique à l'itération conditionnelle[32] est une ***boucle à pré-condition*** (vocabulaire). Elle permet de répéter, zéro ou plusieurs fois, un bloc de code selon l'évaluation, préalable à chaque itération, d'une expression logique servant de condition.

---

30 Pour rappel, en 1.3.2 nous avons vu que les constantes littérales de type caractère, saisies avec des apostrophes, ne sont rien d'autre que des entiers ; elles peuvent donc être utilisées ici.

31 Un tel bloc permet notamment de lever des ambiguïtés en cas de déclaration de variables à cet endroit.

32 Cette notion a été étudiée en détail dans la matière S1-ALR de l'ENIB.

```
int v1=10, v2=20;
while(v1>0)
  {
  --v1;
  v2+=5;
  }
printf("v1=%d v2=%d",v1,v2); // v1=0 v2=70
```

Remarquez que, puisque la condition est évaluée avant chaque itération, il est possible que le corps de la boucle ne soit jamais exécuté (si la condition est fausse dès sa première évaluation). Tout comme pour l'instruction *`if`*, les parenthèses englobant la condition après le mot-clef *`while`* sont obligatoires. En revanche, les accolades marquent simplement un bloc d'instructions ; si une seule instruction est nécessaire, celle-ci n'a pas besoin de figurer dans un bloc. Toutefois, il est recommandé de faire usage d'un bloc si cela facilite la lecture du code.

(http://en.cppreference.com/w/c/language/while)

L'instruction *`do-while`* : la boucle à post-condition

Une autre variante pour provoquer la répétition d'une portion de code consiste en une ***boucle à post-condition*** (vocabulaire). Elle permet de répéter, une ou plusieurs fois, un bloc de code selon l'évaluation, postérieure à chaque itération, d'une expression logique servant de condition.

```
int v1=10, v2=20;
do
  {
  --v1;
  v2+=5;
  } while(v1>0);
printf("v1=%d v2=%d",v1,v2); // v1=0 v2=70
```

Remarquez que, puisque la condition est évaluée après chaque itération, le corps de la boucle est au moins exécuté une fois (même si la condition est fausse dès sa première évaluation). Tout comme pour les instructions *`if`* et *`while`*, les parenthèses englobant la condition après le mot-clef *`while`* sont obligatoires. En revanche, les accolades marquent simplement un bloc d'instructions ; si une seule instruction est nécessaire, celle-ci n'a pas besoin de figurer dans un bloc. Toutefois, il est recommandé de faire usage d'un bloc si cela facilite la lecture du code. Contrairement à l'instruction *`while`*, l'instruction *`do-while`* est terminée par un point-virgule.

L'exemple proposé ici donne un résultat identique à celui de l'instruction *`while`* ; cependant, si la variable *`v1`* avait été initialisée à *`0`*, la boucle *`while`* aurait produit *`v1=0`* et *`v2=20`* alors que la boucle *`do-while`* aurait produit *`v1=-1`* et *`v2=25`*. Il est en pratique assez rare d'avoir besoin d'une boucle à post-condition ; son principal intérêt est d'éviter d'avoir à recopier à l'identique des instructions avant la boucle et dans le corps de la boucle[33].

```
int i=0, result;                  // int i=0, result=doSomething(i);
do                                // while(result>0)
  {                               //   {
  result=doSomething(i);          //   ++i;
  ++i;                            //   result=doSomething(i);
  } while(result>0);              //   }
```

Sur cet exemple, la reformulation en commentaires requiert une duplication de l'expression qui est affectée à *`result`* et nécessite une attention particulière quant à l'incrémentation de l'indice ; l'utilisation de l'instruction *`do-while`* est donc préférable dans ce cas.

(http://en.cppreference.com/w/c/language/do)

33 La duplication de code est une très mauvaise pratique en général puisqu'elle est source d'erreur. En particulier, quand des modifications sont apportées il faut s'assurer de les répéter à l'identique à chaque endroit.

L'instruction *for* : la boucle structurée

La boucle la plus emblématique du langage C est sans conteste l'instruction *for*. Si sa forme est particulièrement bien adaptée à la notion de ***boucle avec compteur*** (vocabulaire), elle offre néanmoins une expressivité toute aussi vaste que la boucle *while*, tout en imposant une organisation plus structurée.

```
for( instruction_initiale ; condition ; instruction_itérative )    // instruction_initiale;
  {                                                                // while( condition )
  /* corps de la boucle */                                        //   {
  }                                                                //   /* corps de la boucle */
                                                                   //   instruction_itérative;
                                                                   //   }
```

La reformulation en commentaires illustre l'interprétation de la structure de la boucle *for*. L'instruction initiale n'est exécutée qu'une seule fois à l'abord de la boucle. La condition est évaluée au début de chaque itération ; c'est donc une pré-condition. L'instruction itérative est exécutée à chaque itération mais après le corps de la boucle ; elle précède juste la prochaine évaluation de la condition.

Les parenthèses englobant ce qui suit le mot-clef *for* sont obligatoires. Elles renferment trois parties optionnelles qui sont séparées par deux point-virgules obligatoires. En revanche, les accolades marquent simplement un bloc d'instructions ; si une seule instruction est nécessaire, celle-ci n'a pas besoin de figurer dans un bloc. Toutefois, il est recommandé de faire usage d'un bloc si cela facilite la lecture du code. Si les instructions initiale ou itérative sont omises, elles n'ont simplement aucun effet, mais s'il n'y a aucune condition, alors celle-ci est considérée comme toujours validée (boucle infinie).

(http://en.cppreference.com/w/c/language/for)

Un usage typique consiste à réaliser une simple boucle avec compteur :

```
for(int i=0;i<10;++i)
  { printf("%d ",i); } // 0 1 2 3 4 5 6 7 8 9
printf("\n");
```

L'intérêt de cette formulation par rapport à la boucle *while* équivalente repose sur le fait que tout le contrôle du compteur tient dans l'en-tête de la boucle *for*, isolé du corps (qui se contente d'y faire référence[34]). La variable qui sert de compteur est d'ailleurs locale à cette boucle, elle ne pollue donc pas l'espace de noms du bloc englobant qui peut éventuellement réutiliser ce même nom pour un tout autre usage. Cette déclaration locale n'est cependant pas imposée ; l'en-tête de la boucle est autorisé à manipuler une variable existante :

```
int i;
for(i=0;i<4;++i ) { printf("< ");  } printf("\n"); // < < < <
for(   ;i<8;i+=2) { printf(">> "); } printf("\n"); // >> >>
printf("i=%d\n",i);                                // i=8
```

qui conserve alors sa dernière valeur après la boucle, comme dans le cas d'une boucle *while*.

Il est également possible de manipuler plusieurs variables à la fois dans l'en-tête de la boucle *for*, mais cette écriture n'est pas très courante. Remarquez également que rien n'impose l'usage d'un compteur pour le contrôle d'une telle boucle :

```
for(double d=0.5;d>1e-3;d*=d) { printf("%g ",d); } printf("\n"); // 0.5 0.25 0.0625 0.00390625
for(;;) { printf("stuck here, forever!\n"); } // L'éternité, c'est long... surtout vers la fin.
```

C'est justement ce qui fait la très grande versatilité de cette forme de boucle ; elle est effectivement utilisée à tout propos en langage C, que ce soit pour compter, calculer, parcourir des structures de données...

34 Il est néanmoins possible de modifier le compteur depuis le corps de la boucle mais ceci est généralement confus.

Que nous ayons affaire à une instruction *while*, *do-while* ou *for*, l'expression logique servant de condition à cette boucle peut être évaluée à de multiples reprises, ce qui peut avoir un coût conséquent en temps d'exécution. Si nous savons que, dans le problème spécifiquement traité, de telles évaluations successives font intervenir des expressions qui produisent le même résultat à chaque itération, alors nous avons tout intérêt à en mémoriser la valeur avant d'envisager les multiples itérations de la boucle.

```
for(int i=0;i<chooseIterationCount();++i)
  { doSomething(i); }
```

Si dans le contexte très spécifique de l'application concernée par cet algorithme nous sommes certains que l'évaluation de la fonction *chooseIterationCount()* donne inexorablement le même résultat, alors nous pouvons reformuler la boucle de cette façon :

```
for(int i=0,count=chooseIterationCount();i<count;++i)
  { doSomething(i); }
```

En revanche, si le corps de la boucle est susceptible de produire un effet de bord (par la fonction *doSomething()* par exemple) qui puisse influer sur le prochain résultat de la fonction *chooseIterationCount()*, nous nous garderons bien d'effectuer une telle transformation. Il n'y a pas de règle systématique en la matière ; seule la bonne connaissance du problème par le programmeur permet de faire un choix raisonné.

Remarquez que, pour cette reformulation, nous avons réutilisé dans l'instruction initiale de la boucle *for* la notation qui déclare plusieurs variables du même type (voir en 1.3.1).

Les instructions *return*, *break*, *continue* et *goto* : les branchements inconditionnels

Nous avons déjà rencontré l'instruction *return* lorsque nous nous sommes intéressés aux fonctions. En effet, puisqu'une fonction peut renvoyer un résultat, celui-ci est fourni par l'instruction *return* suivie d'une expression dont le résultat est d'un type compatible avec le type de retour indiqué dans le prototype de cette fonction. Ce mot-clef peut également être utilisé seul dans le cas où le type de retour de la fonction est *void*.

```
double
compute(int i, double x)
{
printf("i is %d and x is %g\n",i,x);                // i is 8 and x is 12.5
return x-i; // terminer la fonction ici en renvoyant un résultat de type double
printf("not displayed\n");
}

void
testCompute(void)
{
printf("compute(8,12.5) gave %g\n",compute(8,12.5)); //  compute(8,12.5) gave 4.5
return; // terminer la fonction ici sans renvoyer de résultat (type de retour void)
printf("not displayed\n");
}
```

Remarquez que l'usage de l'instruction *return*, avec ou sans valeur de retour, met immédiatement fin à l'exécution de la fonction courante et retourne à la fonction appelante sans que les instructions suivantes ne soient exécutées. C'est également le cas si cette instruction est exécutée depuis l'intérieur d'une alternative ou d'une boucle.

(http://en.cppreference.com/w/c/language/return)

Lors de la présentation de l'alternative multiple, l'instruction *break* a été décrite comme provoquant la sortie immédiate du corps d'une instruction *switch-case*. Ce mot-clef conserve un effet similaire lorsqu'il est employé dans le corps d'une boucle (quelle qu'elle soit) : cette instruction provoque un branchement inconditionnel vers la sortie de la boucle.

```
int data=obtainAnyData(), found=0;
for(int i=0;i<100;++i)             // pour au pire cent valeurs successives
  {
  if(!(i%data))                    // à chaque fois qu'elle est multiple de data
    {
    printf("%d\n",i);              // nous l'affichons
    if(++found==5) { break; }      // nous quittons la boucle après cinq multiples
    }
  }
printf("found %d multiples of %d\n",found,data);
```

Dans cet exemple, nous savons que nous parcourrons la boucle au plus cent fois. Cependant, en fonction des données rencontrées, nous pouvons être amenés à la terminer plus tôt grâce à l'instruction *break*.

Si plusieurs boucles sont imbriquées, seule celle qui englobe le plus directement l'instruction *break* sera concernée[35].

(http://en.cppreference.com/w/c/language/break)

Il existe un autre moyen pour influer sur le déroulement d'une boucle. L'instruction *continue*, utilisée dans le corps d'une boucle provoque un branchement inconditionnel vers la fin de ce corps (sans le quitter). S'il s'agit d'une boucle *while* ou *do-while*, cela provoque directement l'évaluation de la condition pour envisager la prochaine itération ; dans le cas d'une boule *for*, l'instruction d'itération est tout de même exécutée avant la nouvelle évaluation de la condition[36]. L'instruction *continue* trouve son principal intérêt pratique dans une boucle dont le corps est constitué d'une succession d'alternatives qui gèrent chacune un cas exclusif de tous les suivants :

```
for( ... ; ... ; ... )
  {
  // ... obtenir des informations ...
  // ... préparer la condition 1 ...
  if( ... condition 1 ...)
    {
    // ... traiter la condition 1 ...
    continue; // pas la peine d'aller plus loin dans le corps de la boucle
    }
  // ... préparer la condition 2 ...
  if( ... condition 2 ...)
    {
    // ... traiter la condition 2 ...
    continue; // pas la peine d'aller plus loin dans le corps de la boucle
    }
  // ... ainsi de suite ...
  }
```

Sans cette instruction, il aurait été nécessaire d'imbriquer très profondément de multiples instructions *if-else*, ce qui aurait nuit au bout du compte à la lisibilité de la boucle.

(http://en.cppreference.com/w/c/language/continue)

Il y a infiniment peu de raisons d'utiliser l'instruction *goto* puisqu'elle nuit énormément à la clarté d'un algorithme. En effet, elle provoque un branchement vers une étiquette placée à un endroit quelconque au sein de la même fonction, ce qui bouleverse complètement la logique de contrôle des boucles et des alternatives de l'algorithme concerné. Une des très rares situations dans laquelle cette instruction peut malgré tout être utile concerne la sortie directe,

35 De la même façon, si une alternative multiple figure dans le corps d'une boucle, l'instruction *break* rencontrée dans le corps de l'instruction *switch-case* provoquera la sortie de celle-ci mais pas de la boucle.

36 C'est sur ce point que la reformulation d'une boucle *for* par une boucle *while* n'est pas tout à fait équivalente.

suite à la détection d'une anomalie par exemple, de plusieurs boucles imbriquées (impossible avec l'instruction *break*) :

```
for( ... ; ... ; ... )        // boucle 1
  {
  // ...
  for( ... ; ... ; ... )      // boucle 2
    {
    // ...
    if(somethingWentWrong())
      { goto emergencyExit; } // sortie directe
    // ...
    }
  // ...
  }
emergencyExit: // étiquette pour la sortie
printf("done\n");
```

```
// for( ... ; ... ; ... )          // boucle 1
//   {
//   // ...
//   bool breakout=false;
//   for( ... ; ... ; ... )        // boucle 2
//     {
//     // ...
//     if(somethingWentWrong())
//       { breakout=true; break; } // sortie 2
//     // ...
//     }
//   if(breakout) { break; }       // sortie 1
//   // ...
//   }
// printf("done\n");
```

Dans ce cas très particulier, la solution reposant sur l'instruction *goto* exprime plus directement l'intention du programmeur que la version alternative donnée en commentaires. Cette dernière implique en effet la combinaison de multiples instructions *break* associées à une variable que l'on réexamine pour quitter les boucles l'une après l'autre (ce serait encore bien plus confus avec plus de boucles imbriquées).

(http://en.cppreference.com/w/c/language/goto)

1.5. Résumé

L'objectif de ce premier cours était, d'une part, de situer le langage C dans le contexte des langages de programmation en rappelant qu'il permet un contrôle très fin des détails d'exécution des calculs qui sont effectivement réalisés par une machine informatique. D'autre part, il s'agissait de présenter les principes élémentaires et les notations associées permettant de formuler des programmes (très simples dans un premier temps) avec ce langage.

Il est important de maîtriser le vocabulaire de base, en langage courant, servant à décrire les notions abordées ici ; cela conditionne en effet la bonne compréhension mutuelle des ingénieurs et des développeurs qui interagissent autour de ces thèmes. Il est également essentiel de connaître par cœur la forme des constructions du langage (les mots-clefs, la ponctuation, la syntaxe, les règles d'écriture...) sans quoi aucune mise en œuvre ne pourra être envisagée. Beaucoup de ces notions et de ces constructions sont proches de celles qui ont été acquises en suivant la matière S1-ALR de l'ENIB.

Ces connaissances de base serviront de support pour aborder des notions plus spécifiques du langage C. Elles permettront notamment d'envisager des séances pratiques mettant en œuvre des réalisations à des fins d'expérimentation.

2. L01_Build : Organisation et fabrication d'un programme

Motivation :

Historiquement, le langage C servait à faire des applications complètes. Même si cette pratique a quelque peu diminué au profit d'autres langages, il est encore très fréquent d'avoir recours à ce langage pour réaliser des bibliothèques de fonctionnalités qui sont invoquées depuis des langages de plus haut niveau lorsque ceux-ci souffrent de limitations en performances.

Qu'il s'agisse d'une application complète ou d'une bibliothèque de fonctionnalités, le développement se découpe généralement en de multiples parties consacrées à des aspects distincts du problème globalement abordé. La matière S2-IPI de l'ENIB vous a notamment encouragé à un tel découpage.

Ce sujet propose d'introduire par étapes successives, en en justifiant à chaque fois la raison par un exemple, l'ensemble des bonnes pratiques qui devront désormais être suivies pour l'organisation et la compilation d'un tel projet en langage C.

Consignes :

Le travail demandé ici est extrêmement guidé et contient énormément d'explications puisque vous n'êtes pas censé avoir pour l'instant suffisamment de connaissances en langage C pour faire preuve d'initiative sur ce sujet. Veilliez donc à suivre scrupuleusement chacune des étapes en vous assurant à chaque fois d'avoir bien observé et compris ce qui vous est décrit.

2.1. Un premier programme en langage C

Ouvrez un terminal et grâce à la ligne de commande du *shell*, placez-vous dans le répertoire de travail choisi pour ce sujet. Saisissez par exemple[37] :

```
$ mkdir S3PRC_L01_Build ↲
$ cd S3PRC_L01_Build ↲
```

2.1.1. Rédaction du code source

Ouvrez votre éditeur de texte préféré pour rédiger le contenu d'un nouveau fichier nommé `prog01.c`, par exemple :

```
$ gedit prog01.c & ↲
```

Dans celui-ci saisissez (par copier/coller) le texte suivant :

```
int                          // cette fonction doit renvoyer un entier
main(void)                   // définition d'une fonction main() sans paramètre
{
printf("entering main()\n"); // appel d'une fonction printf() avec une chaîne littérale

printf("leaving main()\n");  // un second appel
return 0;                    // voici l'entier que cette fonction doit renvoyer
}
```

37 Les symboles `$` et ↲ représentent respectivement l'invitation à la saisie (le *prompt* du *shell*, qui est déjà affiché dans le terminal) et la validation de la ligne (touche Return/Enter).

Sauvegardez alors le contenu de l'éditeur ; le fichier source `prog01.c` contient désormais le texte saisi. Commençons par en donner une description succincte.

Les portions de lignes qui contiennent le symbole `//` (double-slash) sont des commentaires qui permettent d'ignorer le texte qui s'étend de ce symbole jusqu'à à la fin de la ligne (similaire à `#` que vous connaissez déjà en langage Python). De la même façon, les lignes vides n'ont aucun effet sur le langage ; elle ne sont là que pour la lisibilité.

N'oubliez pas : vous ne programmez pas que pour la machine ! Vous programmez surtout à l'intention d'êtres humains (dont vous-même) qui vont réutiliser, maintenir et étendre votre code ; la lisibilité, l'indentation et les commentaires facilitent ces opérations.

La construction de la forme `int main(void)` désigne une fonction nommée `main` qui n'attend aucun paramètre (signifié par le mot-clef `void`, contrairement à Python ou C++ où des parenthèses vides suffisent), et qui renvoie comme résultat un entier (mot-clef `int`, étudié au chapitre 1). Cette écriture qui consiste à indiquer le type de la valeur renvoyée, le nom et les paramètres d'une fonction est désignée par le terme ***prototype*** (terme de vocabulaire à connaître) ; le prototype d'une fonction est une information fondamentale sur laquelle nous aurons l'occasion de revenir d'ici peu.

Au delà du prototype de cette fonction, nous trouvons une paire de symboles `{ }` (accolades). Un tel bloc contient une séquence d'***instructions*** (vocabulaire), chacune d'elles étant terminée par le symbole `;` (point-virgule) qui est obligatoire. Ce ***bloc*** (vocabulaire) d'instructions qui est joint au prototype matérialise le corps de la fonction c'est à dire l'algorithme que doit exécuter cette fonction lorsqu'elle est appelée.

Contrairement à Python, ce n'est aucunement l'indentation qui marque les blocs d'instructions mais une telle paire d'accolades ; nous pourrions avoir le loisir de disposer les instructions de manière fantaisiste mais il ne faut absolument pas le faire car cela nuirait à la lisibilité. Remarquez qu'ici les instructions et les accolades sont parfaitement alignées ; il s'agit d'un style d'indentation particulier mais il en existe bien d'autres, l'important étant de rester cohérent dans un style une fois qu'on l'a choisi.

Les deux premières instructions du bloc précédent sont des appels à une fonction `printf()` supposée existante et pouvant recevoir en paramètre une chaîne littérale. Une telle chaîne littérale se reconnaît aux symboles `"` (guillemets) qui l'encadrent (existe aussi en Python, vous devez notamment reconnaître le caractère spécial `\n` (backslash-n) qui matérialise un passage à la ligne).

La dernière instruction utilise le mot-clef `return` (comme en Python) afin de fournir la valeur de retour attendue par la fonction : ici un entier (type de retour `int` dans le prototype) valant `0`.

Comme le suggère le nom de la fonction `main()` définie ici, il s'agit de la fonction principale du programme, c'est à dire celle qui est automatiquement appelée quand le programme démarre ; lorsque cette fonction se termine, le programme est également terminé.

Notez que nous n'avons pas le choix quant à son prototype (à une variante près qui sera vue au chapitre 5) ; même si pour l'instant nous ne faisons apparemment aucun usage de l'entier que nous lui faisons renvoyer, celui-ci est obligatoire.

(http://en.cppreference.com/w/c/language/main_function)

2.1.2. Compilation du programme

Contrairement à un programme Python, pour lequel il suffit d'invoquer l'interpréteur en lui indiquant le fichier de code source, un programme en langage C ne peut pas être directement exécuté. Il faut passer par une phase intermédiaire qui traduit ce code source en un format exécutable que le système d'exploitation et la machine informatique comprennent : il s'agit de la ***compilation*** (vocabulaire). Voici une commande qui vise à compiler notre programme :

```
$ gcc prog01.c -o prog01 ↲
```

Le premier mot de cette ligne, `gcc`, représente le nom du ***compilateur*** (vocabulaire) de langage C installé sur la machine. Nous reconnaissons à la suite le nom du programme C que

nous souhaitons lui faire compiler, `prog01.c`, et enfin l'option `-o` accompagnée du nom du programme compilé, `prog01`, que nous souhaitons obtenir[38].

La commande suivante permet de consulter la documentation en ligne (le manuel) décrivant les multiples options de ce compilateur :

```
$ man gcc ↲
```

(c'est très long, monter/descendre avec les flèches, quitter avec la touche `q`)

La compilation de notre programme a produit un message d'avertissement (***warning***, vocabulaire) pour nous signaler une erreur potentielle. Il est à noter que, pour des raisons historiques, le langage C est très permissif : les compilateurs modernes indiquent par des avertissements des formulations qui en toute rigueur devraient être considérées comme des erreurs empêchant la compilation du programme ! Celles-ci ne sont néanmoins pas bloquantes car elles correspondent à des formulations qui étaient autorisées dans les toutes premières versions du langage ; il s'agit de préserver la compatibilité descendante, c'est à dire d'autoriser malgré tout la compilation avec un compilateur moderne d'un code qui avait été rédigé pour un compilateur historique. Lorsqu'il est question comme ici de rédiger à partir de rien un nouveau code, ce comportement est extrêmement risqué ; ignorer un tel avertissement est une très mauvaise pratique ! Nous devons donc chercher à le faire disparaître.

Analysons alors le message produit. Il indique que le problème est détecté dans la fonction `main()` du fichier `prog01.c` ; et plus précisément à une ligne dont le numéro est fourni (affichez les numéros de lignes dans votre éditeur de texte pour la repérer).

Le message évoque une déclaration implicite de la fonction `printf()` ; en effet la ligne de code en question contient bien un appel à cette fonction.

Cette maladresse met en évidence un point fondamental du langage C : ***tout ce que nous utilisons dans une formulation doit avoir été préalablement déclaré*** (qu'il s'agisse d'une variable, d'une fonction, d'un type...). Le langage C ne connaît que les mots-clefs et la ponctuation qui interviennent dans sa grammaire, et dans le cas présent, le compilateur ne peut raisonnablement envisager de traduire un appel à la fonction `printf()` s'il ne sait pas quels sont les paramètres qu'elle attend ni quel est le type du résultat qu'elle doit renvoyer.

La déclaration d'une fonction revient ainsi à écrire son ***prototype***.

Toutefois, nous n'allons pas écrire nous-mêmes la fonction `printf()` puisqu'elle fait partie de la ***bibliothèque standard*** (vocabulaire) du langage C. Il s'agit d'un ensemble de fonctionnalités (des fonctions notamment) déjà compilées et livrées telles quelles lors de l'installation du compilateur sur le système d'exploitation. Lorsque notre compilateur compile notre code source, il complète nos fonctions avec celles de la bibliothèque standard (comme si nous les avions nous-mêmes écrites et déjà compilées). Notre code source doit donc avoir connaissance du prototype de ces fonctions pour envisager d'y faire appel (transmettre le bon nombre de paramètres du bon type, recevoir un résultat du bon type).

Ajoutez alors la ligne suivant avant la définition de votre fonction `main()` :

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

Remarquez que la ligne commence par le symbole `#` (dièse) ; ceci matérialise l'usage d'une ***directive*** (vocabulaire) du ***préprocesseur*** (vocabulaire). Il ne s'agit pas du langage C à proprement parler mais d'un autre langage bien plus rudimentaire dont le propos est de modifier à la volée le code source que le compilateur analysera effectivement. Lorsque nous invoquons le compilateur (par notre commande `gcc` par exemple), celui-ci invoque pour nous le préprocesseur et analyse enfin le code en langage C résultant de cette transformation préalable.

(http://en.cppreference.com/w/c/preprocessor)

Dans le cas présent, il s'agit d'inclure le contenu du fichier `stdio.h` à l'endroit même de cette directive comme si nous l'avions ouvert dans un éditeur de texte, ainsi que tous ceux qu'il

38 Sans cette option, le nom du fichier exécutable choisit par défaut serait `a.out`.

inclut lui même, et avions procédé au copier/coller de toutes les lignes vers le code source de notre programme. Pour nous en rendre compte, utilisons temporairement la commande suivante qui demande au compilateur de n'exécuter que la phase de préprocesseur et d'en afficher le résultat :

```
$ gcc -E prog01.c ↲
```

Vous devriez voir apparaître dans le terminal d'innombrables lignes de code C (difficilement compréhensibles pour l'instant) et enfin le code source de la fonction `main()` tel que vous l'aviez préalablement saisi ; c'est au final ce que voit le compilateur du langage C lorsqu'il analyse votre code source dans le but de le transformer en code exécutable.

De la même façon que pour la bibliothèque standard, le fichier `stdio.h` est installé, ainsi que bien d'autres, dans un emplacement où le compilateur saura le retrouver. Ces fichiers d'extension `.h` (fichiers d'***en-tête***, ***header***, vocabulaire) servent justement à déclarer les fonctionnalités fournies par la bibliothèque standard (et d'autres).

(`http://en.cppreference.com/w/c/io`)

Parmi ces lignes se trouve donc la déclaration (le prototype) de la fonction standard `printf()` qui n'est alors plus inconnue du compilateur, et celui-ci peut enfin s'assurer du fait que l'usage que nous en faisons depuis la fonction `main()` est correct.

Il faut savoir que le compilateur du langage C analyse le code de haut en bas, ce qui fait qu'à chaque ligne analysée il n'a connaissance que de ce qui précède ; il est donc important que l'inclusion du fichier d'en-tête ait eu lieu avant le code qui utilise les fonctionnalités ainsi déclarées.

Dans ces nouvelles conditions, une nouvelle tentative de compilation doit désormais se solder par une réussite, sans message d'erreur ni avertissement :

```
$ gcc prog01.c -o prog01 ↲
```

La commande suivante permet de lister le contenu du répertoire de travail courant :

```
$ ls ↲
```

Vous devez dorénavant y trouver, en plus du fichier de code source, le fichier exécutable qui vient d'être produit.

2.1.3. Exécution du programme

Nous pouvons enfin lancer l'exécution du programme que nous avons rédigé et compilé :

```
$ ./prog01 ↲
```

qui doit provoquer, comme prévu, l'affichage dans le terminal des deux lignes de texte littéral.

Remarquez l'usage du préfixe `./` (point-slash) accolé au nom du programme ; ceci n'a rien à voir avec l'usage du langage C. Lorsque dans le terminal nous saisissons une commande (le nom d'un programme comme `gedit`, `gcc`, `man`, `ls…`), le *shell* qui interprète cette commande sait qu'il peut rechercher un tel programme exécutable dans un ensemble de répertoires prévus à cet effet. Ici nous venons de produire un nouveau programme exécutable qui n'est pas installé dans un de ces répertoires standards ; il est en effet situé dans notre propre répertoire de travail. Le préfixe ainsi utilisé indique au *shell* qu'il trouvera justement ce programme dans le répertoire `.` (point) qui symbolise le répertoire courant[39].

Puisque nous venons de voir que ce programme peut être lancé par le *shell*, c'est le bon moment pour revenir sur l'entier renvoyé par la fonction `main()`. C'est en effet le programme qui a lancé le notre (le *shell* ici) qui peut récupérer cette information[40]. Si juste après avoir exécuté votre programme, vous saisissez la commande *shell* :

---

39 Il est possible de configurer notre environnement pour que le répertoire courant soit automatiquement visité lors de la recherche d'un programme exécutable, et ainsi éviter la saisie du préfixe `./`, mais ceci est généralement considéré comme une mauvaise pratique en terme de sécurité.

40 Généralement la valeur `0` sert à signifier que le programme s'est terminé avec succès et tout autre valeur représente un code d'erreur dont la signification dépend du programme.

```
$ echo $? ↲
```

vous devriez voir apparaître la valeur renvoyée par la fonction `main()` (`0` ici). Pour vous en convaincre, modifiez cette valeur (entre `0` et `255`) dans votre code source, recompilez votre programme, exécutez-le à nouveau et contrôlez la nouvelle valeur obtenue[41].

Nous venons de réaliser, dans sa version minimale, la démarche classique de mise au point d'un programme en langage C :

- édition du code source,
- compilation,
- analyse des messages de compilation,
  - retour éventuel vers l'édition,
- test du programme exécutable obtenu,
  - retour éventuel vers l'édition.

Il faudra désormais toujours procéder de cette façon, même si, comme nous le verrons très prochainement, des détails de réalisation viendront à évoluer.

2.2. Vers une approche modulaire du développement

Il s'agit maintenant d'organiser le code source et le procédé de compilation pour tendre vers une approche dans laquelle, comme indiqué en introduction de ce sujet, une application sera découpée en de multiples ***modules*** (vocabulaire) consacrés à des aspects distincts du problème globalement abordé.

2.2.1. Référence à un module de l'application

Ajoutez, avant chacune des deux instructions de la fonction `main()` qui appellent la fonction `printf()`, cette nouvelle instruction :

```
indent(1);
```

Vous devez reconnaître un appel à une fonction `indent()` à laquelle on transmet un paramètre entier valant `1`.

Vous savez d'ores et déjà que la compilation va afficher un message d'avertissement :

```
$ gcc prog01.c -o prog01 ↲
```

En effet, nous sommes dans le même cas de figure que notre première maladresse : nous cherchons à invoquer une fonction dont le compilateur n'a jamais entendu parler. Seulement cette fois, nous allons réaliser nous-même cette fonction fantaisiste qui ne figure en aucune façon dans la bibliothèque standard.

Cherchons tout d'abord à satisfaire le compilateur en déclarant cette fonction :

```
void
indent(int depth);
```

Cette déclaration doit bien entendu figurer avant la définition de la fonction `main()` qui l'appelle. Remarquez qu'il s'agit bien d'un prototype avec le type de retour (`void` pour aucun), le nom de la fonction et un unique paramètre de type `int` (que nous avons nommé arbitrairement `depth`[42]), ce qui est conforme à l'appel que nous en faisons. Remarquez également que ce prototype est suivi du symbole `;` (point-virgule) et non d'un bloc d'instructions délimité par des accolades ; c'est justement en cela qu'il s'agit d'une ***déclaration*** (vocabulaire) et non d'une ***définition*** (vocabulaire).

Une déclaration constitue en quelque sorte une promesse qui est faite au compilateur pour lui indiquer que quelque part existe une fonction ayant ce nom et correspondant à ce prototype.

41 Dans le terminal, il est confortable de rappeler les dernières commandes sans avoir à les ressaisir. Il suffit d'utiliser les flèches (haut/bas) du clavier et de valider (touche Return/Enter) en ayant éventuellement modifié la commande rappelée.

42 Dans une déclaration de fonction, les noms des paramètres ne sont pas strictement obligatoires (mais les types le sont). Néanmoins c'est une bonne pratique de les préciser car cela renseigne le lecteur sur l'usage qui en sera fait.

Une fois que cette promesse lui est faite, le compilateur peut légitimement valider et traduire des instructions qui cherchent à appeler cette fonction si elles font un bon usage de ses paramètres et de son résultat. Une définition, quant à elle, grâce au bloc d'instructions qu'elle apporte entre ses accolades, fournit le code qui sera exécuté lorsque cette fonction sera appelée[43]. L'intérêt d'une telle distinction vient du fait que, dans une application, toute fonction doit être définie une fois et une seule alors qu'elle peut être invoquée à de multiples reprises depuis de multiples modules. Si nous placions une définition de cette même fonction dans les multiples modules qui ont besoin de l'invoquer, son code compilé figurerait au bout du compte en de multiples exemplaires dans le programme exécutable produit.

Compilons encore notre code source pour observer le comportement du compilateur vis-à-vis de la déclaration que nous venons d'ajouter. Il signale à nouveau un échec mais celui ci est d'une toute autre nature que le précédent. Il n'est plus question d'une maladresse concernant une instruction particulière de notre code, mais il nous rappelle que nous n'avons pas tenu la promesse que nous venons de lui faire ! En effet, la fonction `indent()` que nous avons déclarée n'est définie nulle part alors que la fonction `main()` y fait référence.

Ce type d'erreur concerne l'***édition de liens*** (***link***, vocabulaire). De la même façon que nous avions découvert qu'avant la compilation proprement dite intervenait le préprocesseur, nous constatons à la toute fin l'intervention de l'***éditeur de liens*** (***linker***, vocabulaire). Le rôle de cette phase consiste justement à inspecter tous les modules compilés (il n'y en a qu'un ici pour l'instant) et à s'assurer que toutes les fonctions invoquées ont bien une définition et une seule. Ces définitions peuvent être retrouvées parmi les modules compilés, dans la bibliothèque standard ou éventuellement dans d'autres bibliothèques fournies à la demande.

2.2.2. Ajout d'un module à l'application

Puisque nous travaillons sur l'approche modulaire, nous allons justement réaliser un module supplémentaire qui définit la fonction manquante. Ouvrez votre éditeur de texte préféré pour rédiger le contenu d'un nouveau fichier nommé `module_A.c`, par exemple :

```
$ gedit module_A.c & ↲
```

Dans celui-ci saisissez (par copier/coller) le texte suivant :

```
#include <stdio.h>

void
indent(int depth)
{
if(depth!=0)
  { printf("| "); indent(depth-1); }
}
```

La définition de la fonction `indent()` n'est guère plus compliquée que celle de la fonction `main()` que vous avez déjà réalisée. Nous y retrouvons son prototype déjà décrit lors de sa déclaration ainsi que, parmi ses instructions, des appels de fonctions. Nous retrouvons l'usage de la fonction standard d'affichage de texte `printf()` (qui nécessite l'inclusion du fichier d'en-tête `stdio.h`), ainsi qu'un appel récursif[44] à la fonction `indent()` elle-même. L'instruction `if`, qui est présentée au chapitre 1, a la même signification qu'en Python : le deuxième bloc d'instructions imbriqué dans le premier ne sera exécuté que si le paramètre entier de la fonction n'est pas nul.

Puisque nous disposons maintenant de plusieurs fichiers sources, nous allons légèrement modifier le procédé de fabrication du programme exécutable en ayant recours à la ***compilation séparée*** (vocabulaire).

---

43 Bien entendu, puisque la définition d'une fonction en expose le prototype avant les accolades, elle fait également office de déclaration.

44 Vous avez vu les appels récursifs dans la matière S1-ALR de l'ENIB. Cette fonction s'appelle elle même avec son paramètre entier décrémenté de `1` lorsque ce paramètre n'est pas nul ; lorsqu'elle sera finalement invoquée avec un paramètre valant `0`, elle ne fera rien ce qui mettra fin aux récursions.

Comme le nom le suggère, nous commençons par compiler séparément les différents modules avec ces commandes :

```
$ gcc prog01.c -c ↲
$ gcc module_A.c -c ↲
```

Vous remarquez certainement que l'option `-c` a été ajoutée et que nous ne fournissons plus le nom du fichier exécutable. Effectivement, ces commandes ne cherchent pas à générer le programme exécutable ; l'option `-c` leur dit de s'arrêter à l'étape de la compilation proprement dite sans effectuer l'édition de liens. Le résultat de chacune de ces commandes est un ***fichier objet*** (vocabulaire) qui contient la traduction, en code exécutable par la machine, des fonctions définies dans chacun des fichiers sources, ainsi que des informations à propos des autres fonctions dont ce fichier a besoin. La liste du contenu du répertoire courant :

```
$ ls ↲
```

doit maintenant révéler la présence de ces deux fichiers objets qui utilisent l'extension `.o` en lieu et place de l'extension `.c` de chaque fichier de code source.

Il ne reste plus qu'à les rassembler lors de l'édition de liens :

```
$ gcc prog01.o module_A.o -o prog01 ↲
```

Nous retrouvons enfin le nom du programme exécutable qui doit être généré. Remarquez toutefois que nous invoquons le compilateur `gcc` mais que celui-ci ne fait lors de cette dernière commande aucune compilation à proprement parler. Il n'y a en effet sur sa ligne de commande aucun fichier source en langage C mais uniquement les fichiers objets obtenus précédemment ; il se contente alors de passer directement à la phase d'édition de liens afin de vérifier qu'aucune définition ne manque ni n'est redondante et procéder enfin à la production du programme exécutable[45]. Vous pouvez alors tester ce dernier pour vous assurer qu'il affiche toujours les deux lignes de texte initiales mais dorénavant précédées d'un petit motif textuel matérialisant une indentation de l'affichage (effet de notre fonction `indent()`).

Bien entendu, le principe du découpage modulaire présenté ici paraît bien compliqué pour si peu de code[46] mais il est à la base de la démarche qui est systématiquement adoptée dans tout projet raisonnable (pensez aux centaines de fichiers sources qui constituent les outils LibreOffice ou Firefox). Le point essentiel à retenir concerne la distinction entre la définition et la déclaration d'une fonction. Dans le but d'acquérir à terme un savoir faire exploitable sur des projets de plus grande envergure que ces exercices introductifs, nous nous efforcerons dorénavant d'adopter systématiquement cette démarche même lorsqu'elle semble superflue.

2.3. Démarche de compilation séparée

Il est maintenant question d'acquérir une démarche systématique pour faciliter la mise en œuvre d'un développement modulaire. Dans ce but, nous provoquerons volontairement des incohérences afin d'introduire petit à petit les bonnes pratiques qui permettent de s'en prémunir.

2.3.1. Modification d'un module

Imaginons que nous souhaitions ajouter une fonctionnalité à notre fonction `indent()` afin de choisir le motif textuel qu'elle doit répéter. Il nous faut modifier cette fonction afin qu'elle accepte un nouveau paramètre représentant ce motif et qu'elle s'en serve lors de ses appels à `printf()`.

Modifiez alors le code source du fichier `module_A.c` afin qu'il contienne ceci :

---

45 Les développeurs utilisent couramment le même terme ***compilation*** pour désigner à la fois l'opération qui consiste, comme ici, à générer le fichier objet correspondant à un fichier de code source et aussi pour désigner, comme dans la première version de notre programme, l'obtention du programme exécutable à partir de l'ensemble de ces fichiers sources (en incluant l'édition de lien donc). Le contexte suffit généralement à faire la distinction.

46 Nous aurions pu, par facilité, rédiger tout le code dans un unique fichier source ou bien compiler l'ensemble des fichiers sources dans une seule commande `gcc` mais ceci ne correspond à aucun scénario réaliste pour un ingénieur.

```
#include <stdio.h>

typedef const char *Tag; // déclaration du type Tag

void
indent(Tag tag, // un nouveau paramètre de type Tag
       int depth)
{
if(depth!=0)
  { printf(tag); indent(tag,depth-1); }
}
```

Vous ne connaissez pas encore suffisamment de détails du langage C pour savoir spécifier précisément le type du nouveau paramètre ; nous le désignons par le type *Tag* que nous inventons ici pour le propos de l'exercice (ne cherchez donc pas à comprendre les détails de la construction *typedef*). Recompilez ensuite l'ensemble de l'application :

```
$ gcc prog01.c -c ↲
$ gcc module_A.c -c ↲
$ gcc prog01.o module_A.o -o prog01 ↲
```

L'absence de message vous assure que pour le compilateur et l'éditeur de liens l'ensemble du code semble correct. Testez-le alors en exécutant cette nouvelle version du programme exécutable produit. Il est très probable que le *shell* depuis lequel vous le lancez vous indique un plantage causé par une erreur de segmentation (abordée dans un futur sujet).

2.3.2. Digression : analyse d'un plantage au débogueur

Le précédent problème est l'occasion rêvée pour introduire un nouvel outil qui a toute sa place dans le processus de développement d'une application en langage C : le ***débogueur*** (***debugger***, vocabulaire). La commande *gdb* fait partie des outils livrés dans la chaîne de développement qui accompagne *gcc*. C'est un débogueur qui s'utilise en ligne de commande ; même s'il est très puissant et largement utilisé par un grand nombre de développeurs, il n'est pas très confortable pour débuter. Il existe par ailleurs des interfaces graphiques évoluées qui pilotent cet outil ; elles nécessitent, malgré les apparences, une longue phase d'apprentissage et sont généralement lourdes à utiliser (gros programmes à la réaction lente). Par chance, l'enseignement autour des microprocesseurs à l'ENIB s'appuie énormément sur l'usage d'un débogueur qui facilite les opérations courantes[47]. Nous nous appuierons donc sur cet outil ici pour en découvrir l'usage minimal ; vous le retrouverez lorsque vous étudierez les microprocesseurs.

Un débogueur permet principalement de suivre dans le code source ce qui se déroule durant l'exécution du programme exécutable. Seulement, lors de la compilation, tout le code source disparaît au profit du code exécutable par la machine. Si nous envisageons d'utiliser un débogueur, Il est alors nécessaire d'indiquer au compilateur qu'il doit conserver dans les fichiers objets, et donc dans le programme exécutable, des informations qui permettent de relier les instructions du code exécutable à celles du code source. Il suffit d'ajouter l'option *-g* à chaque commande de compilation :

```
$ gcc prog01.c -c -g ↲
$ gcc module_A.c -c -g ↲
```

L'édition de liens, quant à elle, doit être réalisée comme précédemment :

```
$ gcc prog01.o module_A.o -o prog01 ↲
```

Le programme exécutable ainsi généré contient des informations permettant son débogage :

```
$ tdb prog01 ↲
```

47 L'outil *tdb* est une interface graphique très légère pour *gdb*. Éric Boucharé, enseignant à l'ENIB, la maintient pour qu'elle réponde aux besoins en électronique et nous l'avons adaptée pour l'enseignement de l'informatique.

Cet outil ouvre une interface graphique comprenant deux parties principales : en haut la visualisation du code source précédée de boutons de commandes, en bas une zone de messages suivie d'une ligne de saisie de commandes.

Notre programme est initialement en pause ; la toute première ligne de la fonction `main()` est mise en surbrillance pour indiquer qu'il s'agit de la prochaine ligne de code qui sera exécutée. Nous procéderons à l'investigation selon les étapes suivantes[48] :

- le bouton `Continue [F8]` (triangle) permet de lancer l'exécution du programme,
  - la zone de messages indique que le programme est arrêté car il s'est effectivement planté à cause d'une erreur de segmentation,
- le bouton `Call Stack` (boîtes empilées) affiche dans le zone de messages l'empilement des contextes d'appel depuis la fonction `main()` jusqu'au point de plantage,
  - nous constatons que le plantage s'est révélé dans des fonctions de la bibliothèque standard du langage C invoquées par `printf()` ; bien entendu nous n'y avons pas accès et nous n'y pouvons rien,
- l'usage répété du bouton `Frame Up` (boîtes et flèche montante) permet d'atteindre le contexte d'un appel de fonction qui nous concerne,
  - à chaque appui, un message indique le contexte courant ; nous arrêtons lorsque notre fonction `indent()` est atteinte,
  - à ce moment, la zone de code met en surbrillance la ligne courante dans ce contexte (l'invocation de `printf()`) et le message indique la valeur des paramètres de la fonction `indent()`,
    - le paramètre `depth` a une valeur entière complètement fantaisiste,
    - le paramètre `tag` désigne une zone mémoire inaccessible, que nous transmettons à `printf()`, ce qui explique le plantage,
- un nouvel appui sur le bouton `Frame Up` (boîtes et flèche montante) nous fait atteindre le contexte de la fonction `main()`,
  - la zone de code met en surbrillance la ligne courante dans ce contexte,
    - l'invocation de `indent()` transmet un seul paramètre alors que dans le contexte visité précédemment nous constations que deux étaient attendus ; cela explique la valeur fantaisiste obtenue pour le deuxième paramètre,
    - l'unique paramètre transmis est un entier valant 1 alors que dans le contexte visité précédemment le premier paramètre attendu est une chaîne de caractères ; cela explique l'incohérence qui est à l'origine du plantage,
- nous quittons la session de débogage en fermant simplement la fenêtre de l'outil.

L'utilisation du débogueur était ici purement illustrative car l'incohérence détectée était facilement prévisible. Néanmoins, dans le cas de programmes plus élaborés, le parcours et l'inspection de la pile des contextes d'appels des fonctions lors d'un plantage est une démarche très classique pour rechercher la cause d'un dysfonctionnement.

2.3.3. Cohérence entre déclaration et définition

Bien que nous ayons fini par montrer que l'appel qui est fait à la fonction `indent()` ne correspond pas à ce qui est attendu dans sa définition, il est tout de même très regrettable que cette incohérence soit passée inaperçue lors de la fabrication du programme exécutable. La fonction `main()` s'appuie en effet sur une déclaration de la fonction `indent()` qui n'est plus à jour par rapport à sa nouvelle définition (depuis l'ajout d'un paramètre). Malheureusement, l'édition de liens se contente de vérifier la présence de chaque fonction requise mais n'a aucun moyen de confronter les signatures requises et disponibles.

De la même façon que nous avions eu recours au fichier d'en-tête standard `stdio.h` pour prendre connaissance de la déclaration de la fonction `printf()`, nous allons fournir un fichier d'en-tête qui déclare les fonctionnalités que le `module_A.c` définit.

48 Il est très hasardeux de décrire comme ici le comportement d'un programme incorrect. En effet, à l'opposé d'un programme correct qui doit avoir un comportement prévisible et descriptible avec précision, un programme incorrect a un comportement indéterminé qui peut varier entre les différentes machines et même d'une exécution à une autre.

Ouvrez votre éditeur de texte préféré pour rédiger le contenu d'un nouveau fichier nommé *module_A.h*, par exemple :

```
$ gedit module_A.h & ↲
```

Dans celui-ci saisissez (par copier/coller) le texte suivant :

```
typedef const char *Tag;

void
indent(Tag tag,
       int depth);
```

Il ne s'agit que de la déclaration du type *Tag* déjà vu précédemment et de la déclaration de la fonction *indent()* selon son nouveau prototype.

Dans le fichier *module_A.c* nous devons retirer la construction *typedef* qui déclare le type *Tag* et ajouter au début du fichier l'inclusion de notre fichier d'en-tête :

```
#include "module_A.h"
#include <stdio.h>

void
indent(Tag tag,
       int depth)
{
if(depth!=0)
  { printf(tag); indent(tag,depth-1); }
}
```

De la même façon, dans le fichier *prog01.c* nous devons remplacer la déclaration de la fonction *indent()* par cette même inclusion :

```
#include <stdio.h>
#include "module_A.h"

int
main(void)
{
indent(1); printf("entering main()\n");

indent(1); printf("leaving main()\n");
return 0;
}
```

Remarquez que dans ces deux cas nous encadrons le nom de notre propre fichier d'en-tête par des symboles " (guillemets) alors que pour *stdio.h* nous utilisons < et > (chevrons). Cette distinction repose sur le fait que ces fichiers d'en-tête à inclure sont locaux au projet en cours ou bien sont déjà installés dans les répertoires standards du système d'exploitation[49].

(http://en.cppreference.com/w/c/preprocessor/include)

Sans rien changer d'autre au code source, la compilation du fichier *module_A.c* doit se produire sans encombre. En effet, la déclaration découverte dans le fichier d'en-tête est bien strictement conforme au prototype de la définition fournie un peu plus loin dans le fichier source. En revanche, la compilation du fichier *prog01.c* doit désormais produire des messages d'erreur puisque cette même déclaration ne correspond plus du tout aux appels qui sont exprimés dans la fonction *main()*. Remarquez que si nous avions choisi de conserver l'ancien prototype de la fonction *indent()* pour sa déclaration dans le fichier *module_A.h*, ce serait

49 Ceci a une influence sur la recherche des fichiers à inclure. Les chevrons provoquent la recherche uniquement dans les répertoires standards, alors qu'avec les guillemets elle commence par le répertoire courant et, si le fichier à inclure n'y est pas trouvé, la recherche se poursuit dans les répertoires standards.

alors la compilation de *module_A.c* qui échouerait (incohérence entre la définition et sa déclaration préalable) et celle de *prog01.c* qui réussirait.

Dans le but d'éliminer ces erreurs, nous rectifions alors les deux appels à *indent()* dans la fonction *main()* de façon à ce qu'ils soient à nouveau conformes au prototype de la déclaration :

```
indent("* ",1);
```

Les deux compilations et l'édition de liens doivent désormais se dérouler sans encombre. Le programme exécutable obtenu doit maintenant illustrer l'effet de nos modifications en utilisant le nouveau symbole choisi pour l'indentation des deux lignes de texte affichées.

Par l'usage d'un fichier d'en-tête qui déclare les fonctionnalités d'un module, nous venons d'établir un principe qui assure de manière certaine la cohérence entre les déclarations et les définitions. Il suffit d'inclure systématiquement ce fichier d'en-tête au début du fichier de définition du module qu'il décrit. Il ne reste alors plus qu'à inclure ce même fichier d'en-tête dans les autres modules qui ont besoin de réaliser des appels au module ainsi déclaré et défini. À la moindre incohérence, dans un cas ou dans l'autre, le compilateur nous signale une erreur.

Nous n'y avons pas eu recours dans l'exemple simpliste traité ici, mais il serait tout à fait raisonnable d'inclure d'autres fichiers d'en-tête (standards ou propres à l'application) à l'intérieur de celui que nous venons de produire. En effet, un fichier d'en-tête correctement rédigé doit être autosuffisant, c'est à dire qu'on doit pouvoir l'inclure dans n'importe quel module sans qu'il soit nécessaire de faire précéder cette inclusion d'autres déclarations.

Un bon moyen de s'en assurer consiste à placer l'inclusion de notre propre fichier d'en-tête au tout début du code source du module qu'il décrit. C'est ce que nous avons fait pour *module_A.h* qui figure bien sur la première ligne de *module_A.c*.

2.3.4. Modules interdépendants

Nous poursuivons notre recherche des situations problématiques en ajoutant à notre programme un nouveau module qui dépend du précédent.

Ouvrez votre éditeur de texte préféré pour rédiger le contenu d'un nouveau fichier nommé *module_B.c*, par exemple :

```
$ gedit module_B.c & ↲
```

Dans celui-ci saisissez (par copier/coller) le texte suivant :

```
// ...c'est à vous de compléter le début de ce fichier...

void
doSomething(Tag tag,
            int depth)
{
indent(tag,depth); printf("entering doSomething()\n");
indent(tag,depth); printf("...working hard...\n");
indent(tag,depth); printf("leaving doSomething()\n");
}
```

Nous reconnaissons sans difficulté la définition d'une nouvelle fonction *doSomething()* dont le prototype et le corps ne reposent que sur des éléments déjà décrits. Vous devez vous empresser d'appliquer immédiatement les recommandations précédentes en incluant au tout début de ce fichier l'en-tête *module_B.h* (qui n'existe pas encore) associé à ce module. Pour les appels à *indent()* et *printf()*, il nous est à nouveau nécessaire d'inclure *module_A.h* et *stdio.h*.

Il nous faut désormais fournir le fichier *module_B.h* dont le contenu devra déclarer la fonction *doSomething()* (c'est à vous de le faire en vous inspirant du travail précédent). Seulement, le prototype de cette fonction fait référence au type Tag (pour transmettre le premier paramètre à

indent()) que le compilateur ne connaît pas encore à ce stade. Ajoutez alors, avant la déclaration de *doSomething()*, l'inclusion de *module_A.h*.

Maintenant que notre module est complètement déclaré et défini, nous pouvons en faire usage dans le programme principal. Entre les deux lignes qui affichent des messages dans la fonction *main()*, ajoutez un appel à notre nouvelle fonction.

```
doSomething("| ",2);
```

Vous n'aurez bien entendu pas oublié d'ajouter à cette occasion l'inclusion de *module_B.h* dans *prog01.c*.

La reconstruction complète de votre programme repose désormais sur ces commandes :

```
$ gcc prog01.c -c ↲
$ gcc module_A.c -c ↲
$ gcc module_B.c -c ↲
$ gcc prog01.o module_A.o module_B.o -o prog01 ↲
```

Malheureusement, les compilations de *prog01.c* et *module_B.c* échouent en indiquant que le type *Tag* est déclaré à de multiples reprises[50]. En effet, ces deux fichiers incluent à la fois *module_A.h* et *module_B.h*, mais le fichier d'en-tête *module_B.h* inclut lui-même *module_A.h*, ce qui fait qu'après la phase de préprocesseur, le compilateur voit deux fois les déclarations de *module_A.h*. Pour nous en convaincre, visualisons à nouveau le travail du préprocesseur :

```
$ gcc -E prog01.c ↲
$ gcc -E module_B.c ↲
```

Les déclarations du type *Tag* et de la fonction *indent()* apparaissent bien deux fois dans chacun de ces deux cas.

Une solution potentielle pour tenter de remédier à ce problème serait de s'interdire d'inclure *module_A.h* partout où nous incluons *module_B.h* (puisqu'il le fait pour nous). Cependant, si nous avions affaire à des dizaines de modules dont plusieurs d'entre-eux seraient interdépendants comme ici, les intrications seraient telles que cette démarche serait inapplicable ! Il nous faut une nouvelle fois trouver un procédé systématique qui résout ce problème.

Modifiez simplement le fichier d'en-tête module_A.h de la façon suivante :

```
#ifndef MODULE_A_H
#define MODULE_A_H 1

// ...ici l'ancien contenu du fichier...

#endif // MODULE_A_H
```

et relancez le processus de construction du programme exécutable. Cette fois il doit aboutir avec succès et le programme obtenu doit provoquer l'affichage des lignes de texte attendues.

Comme le laissent deviner les symboles # (dièse) au début de chaque ligne ajoutée, nous utilisons ici les services du préprocesseur. Puisque ce dernier contrôle les lignes de code que le compilateur de langage C voit effectivement, nous lui donnons des consignes pour que le contenu de ce fichier d'en-tête ne soit vu qu'une seule fois par ***unité de compilation*** (***translation unit***, vocabulaire). Ce terme désigne la production d'un unique fichier objet à partir d'un unique module et des multiples fichiers d'en-tête qu'il inclut ; concrètement il s'agit d'une unique invocation du compilateur avec l'option *-c*.

50 Si ce n'est pas le cas, recommencez la compilation en ajoutant les options *-std=c99* *-pedantic* après l'option *-c*. Les différentes versions des compilateurs ne sont pas toutes aussi sévères ; pour produire du code portable il ne faut pas compter sur le comportement tolérant de certains d'entre-eux.

Comme décrit en 1.4.1, le langage du préprocesseur permet de définir des ***macro-instructions*** (ou ***macros***, vocabulaire). C'est le cas ici avec la directive `#define` qui définit la *macro* `MODULE_A_H` avec le contenu `1`. La directive `#ifndef` (contraction de *if not defined*) est appairée avec la directive `#endif` pour exprimer une ***compilation conditionnelle*** (vocabulaire).

Dans ces conditions, si la *macro* `MODULE_A_H` n'est pas définie, alors nous la définissons[51] et les lignes de code enfermées dans cette construction restent visibles du compilateur. En revanche, si cette *macro* est déjà définie, ces mêmes lignes de codes deviennent invisibles. Lorsque le préprocesseur commence son travail, la *macro* en question n'est pas encore définie, donc la première fois qu'il cherchera à inclure ce fichier d'en-tête, son contenu sera intégralement pris en compte ce qui aura également pour effet de définir cette *macro*. Si en poursuivant son travail, le préprocesseur cherche à inclure à nouveau ce même fichier, alors celui-ci semblera n'avoir cette fois aucun contenu. Tout ceci fait qu'au bout du compte le compilateur de langage C ne verra qu'une seule fois les lignes de code constitutives d'un même fichier d'en-tête, même si celui-ci est inclus plusieurs fois au sein d'une unité de compilation.

Bien entendu, ceci suppose que chaque fichier d'en-tête soit associé à un nom de *macro* qui lui est spécifique, sans quoi l'inclusion de l'un d'eux risquerait d'inhiber l'inclusion d'un autre. Une habitude courante consiste à s'inspirer, comme ici, du nom du fichier pour composer le nom de la *macro*[52]. Afin de systématiser cette pratique, veuillez faire une intervention similaire sur le fichier `module_B.h` avec une *macro* nommée `MODULE_B_H`. Relancez enfin tout le processus de construction du programme exécutable pour vous assurer qu'aucune nouvelle erreur ne survient et que le programme reste fonctionnel.

Cette manière de rédiger et d'utiliser les fichiers d'en-tête, pour construire un programme par compilation séparée, est un grand classique qu'il faudra dorénavant appliquer de façon systématique dans toutes vos futures réalisations.

2.3.5. Digression : analyse de l'exécution au débogueur

Maintenant que nous disposons d'un programme reposant sur plusieurs modules interdépendants, nous pouvons revenir à l'usage du débogueur pour inspecter les multiples appels de fonctions dont il est constitué. Avant ceci, il vous faudra bien sûr compiler l'ensemble du code source avec l'option `-g`.

Au lancement de l'outil de débogage :

```
$ tdb prog01 ↲
```

notre programme est initialement en pause ; la toute première ligne de la fonction `main()` est mise en surbrillance pour indiquer qu'il s'agit de la prochaine ligne de code qui sera exécutée. Nous procéderons à l'investigation selon les étapes suivantes :

- un *double-clic* dans la zone d'affichage du code source, juste sur la ligne qui provoque l'appel à la fonction `doSomething(),` place un ***point-d'arrêt*** (*break-point*, vocabulaire) matérialisé par un changement de couleur,
- le bouton `Continue [F8]` (triangle) permet de lancer l'exécution du programme,
  - la première ligne de texte littéral est produite dans la console et la zone de messages indique que le programme est arrêté sur le point-d'arrêt,
- le bouton `Step [F5]` (entrée dans les accolades) permet d'atteindre la prochaine ligne de code dans le programme,
  - nous sommes désormais sur la première ligne de la fonction `doSomething()` et la zone de messages nous indique la valeur de ses paramètres,
- de nouveaux appuis sur ce même bouton nous font progresser ***pas-à-pas*** (*step-by-step*, vocabulaire) dans l'exécution,
  - à chaque fois, utilisez le bouton `Call Stack` pour visualiser la pile des appels de fonctions et leurs paramètres dans la zone de messages,

51 Le fait de choisir `1` comme contenu pour cette *macro* n'a aucune influence ici ; ce contenu est même facultatif.

52 Bien que rien ne l'impose, il est extrêmement courant de n'utiliser que des majuscules dans un nom de *macro*.

        - vous pouvez même utiliser les boutons *Frame Up* et *Frame Down* pour inspecter la pile d'appels alors que l'exécution reste arrêtée au même point,
        - veuillez remarquer qu'une même fonction appelée récursivement a bien, à un moment donnée, plusieurs exemplaires de ses paramètres avec des valeurs distinctes pour chacune des invocations,
    - après la dernière invocation de *indent()*, pour laquelle le paramètre *depth* vaut *0*, vous serez revenu à une nouvelle ligne de la fonction *doSomething()* et la deuxième ligne de texte littéral de votre programme aura été produite dans la console,
- le bouton *Next [F6]* (saut au dessus des accolades) permet d'atteindre la prochaine ligne de code sans s'arrêter dans une fonction appelée,
    - la troisième ligne de texte littéral est produite dans la console et nous sommes juste avant le troisième appel à *indent()* dans *doSomething()*,
- le bouton *Finish [F7]* (sortie des accolades) permet de s'arrêter juste après la sortie de la fonction en cours,
    - la quatrième ligne de texte littéral est produite dans la console et nous sommes revenus dans la fonction *main()*,
- le bouton *Continue [F8]* relance l'exécution,
    - la cinquième ligne de texte littéral est produite dans la console et la zone de messages nous indique que la fin du programme est atteinte,
- nous quittons la session de débogage en fermant simplement la fenêtre de l'outil.

L'utilisation qui a été faite du débogueur ici visait à illustrer la démarche très classique qui consiste à atteindre quelques points intéressants d'un programme pour poursuivre progressivement son exécution en s'interrogeant à chaque fois sur les fonctions qui sont appelées et les paramètres qui leur sont transmis. Ceci peut aider à la mise au point d'un programme en recherchant les maladresses qui peuvent être à l'origine d'un fonctionnement inattendu.

2.4. Automatisation de la construction

Ces expérimentations autour de l'organisation et de la rédaction du code source nous ont conduits à relancer à plusieurs reprises le procédé de construction du programme exécutable. Il s'agissait à chaque fois d'invoquer les multiples commandes de compilation (en changeant parfois les options, pour le débogage par exemple) ainsi que celle chargée de l'édition de liens. Ce besoin reste très fréquent lorsque nous développons une application ; en effet il est recommandé de s'assurer régulièrement que les modifications apportées autorisent toujours une construction et une exécution correcte du programme.

Il pourrait sembler confortable d'inscrire une fois pour toutes l'intégralité de cette séquence de commandes dans un fichier *script* que le *shell* pourrait exécuter à notre demande. Nous ferions ainsi l'économie de multiples rappels de commandes ; celle du *script shell* suffirait à elle seule.

Dans le cas présent, cette solution serait suffisante puisque nous ne manipulons que très peu de fichiers. En revanche dans un projet de grande ampleur, la compilation de l'ensemble du code demande une durée non négligeable pendant laquelle le développeur est obligé d'attendre.

2.4.1. Un fichier makefile très spécifique

Pour répondre au besoin évoqué, il existe un outil qui fait partie intégrante de toute chaîne de développement (avec le compilateur, le débogueur...) : le programme *make*. Sans être spécifique au langage C, il permet d'automatiser tout processus qui consiste à générer des fichiers à partir d'autres ; c'est justement le cas de la compilation et de l'édition de lien.

Lorsqu'on invoque ce programme, il analyse le contenu du fichier nommé *makefile*[53] (vocabulaire) situé dans le répertoire courant, afin d'y découvrir des relations de dépendance entre des fichiers (le programme exécutable, les fichiers objets, les fichiers sources, les fichiers d'en-tête...) ainsi que les commandes qui permettront de les résoudre (compilation, édition de liens...).

53 Il peut aussi s'agir de *Makefile* ou encore de *GNUmakefile* selon la version de ce programme.

Ouvrez votre éditeur de texte préféré pour rédiger le contenu d'un nouveau fichier nommé *makefile*, par exemple :

```
$ gedit makefile & ↲
```

Dans celui-ci commencez par saisir le texte suivant :

```
# édition de liens
prog01 : prog01.o module_A.o module_B.o
<~TAB~> gcc prog01.o module_A.o module_B.o -o prog01
```

Le symbole # (dièse) représente le début d'un commentaire qui s'étend jusqu'à la fin de la ligne (comme en Python). La ligne suivante représente une ***règle*** (vocabulaire). Elle s'interprète de la façon suivante : la ***cible*** (vocabulaire) qui est à gauche du symbole *:* (deux-points) peut être produite à partir des ***prérequis*** (vocabulaire) qui sont à droite de ce même symbole ; nous reconnaissons effectivement ici le programme exécutable qui peut être produit à partir des trois fichiers objets. La ligne qui suit immédiatement cette relation de dépendance débute par l'usage de la touche de tabulation. Ceci indique qu'il s'agit de la ***commande*** (vocabulaire) qui permet de résoudre la dépendance immédiatement précédente pour fabriquer sa cible[54] ; nous reconnaissons effectivement ici la commande d'édition de liens que nous utilisions jusqu'alors. Le programme *make*, à la lecture de ce fichier *makefile*, saura qu'à chaque fois qu'il faudra produire le programme exécutable il suffira d'invoquer cette commande.

Seulement, pour l'instant rien n'indique comment obtenir les fichiers objets en question. C'est donc à vous de compléter ce fichier *makefile* en y ajoutant à la suite trois blocs règle/commande semblables au précédent. Chacun d'eux devra indiquer qu'un fichier objet *.o* particulier s'obtient à partir du fichier de code source *.c* correspondant en invoquant la commande de compilation (avec l'option *-c*) adéquate. Vous pouvez laisser des lignes vides entre ces blocs pour faciliter la lecture.

Complétez enfin ce fichier *makefile* en y ajoutant, toujours à la suite, le bloc suivant :

```
# nettoyage
clean :
<~TAB~> rm -f prog01 *.o
```

Cette règle est quelque peu particulière puisque la cible ne correspond à aucun fichier et qu'il n'y a aucun prérequis. La commande, quant à elle, repose sur la commande *rm* du *shell* qui sert à supprimer des fichiers ; ici le programme exécutable et les fichiers objets. Cette construction est grand un classique du genre[55] ; elle sert à faire du ménage parmi les fichiers en supprimant tous ceux qui peuvent être générés automatiquement.

Désormais, lorsque vous saisissez dans le terminal la commande :

```
$ make clean ↲
```

ce programme va chercher à fabriquer la cible *clean* explicitement indiquée. Comme aucun fichier ayant ce nom n'existe et puisque le fichier *makefile* indique qu'elle ne dépend d'aucun prérequis, la commande de suppression des fichiers sera exécutée de manière inconditionnelle.

Si maintenant vous saisissez dans le terminal la commande :

```
$ make ↲
```

ce programme cherchera à fabriquer la première cible[56] rencontrée dans le fichier *makefile* ; il s'agit ici de *prog01*. Les prérequis sont alors analysés mais les fichiers objets en question n'existent pas, ce qui rend impossible l'exécution immédiate de la commande d'édition de liens. Heureusement, vous avez ajouté les trois règles concernant ces cibles ; le programme *make* va donc chercher à les fabriquer. Chacune d'elles a pour prérequis un fichier de code source qui est bien présent ; la commande de compilation peut donc être exécutée pour chacun d'eux, ce qui produit les trois fichiers objets attendus. Le programme *make* peut enfin exécuter la commande d'édition de liens pour produire sa cible principale *prog01*.

---

54 Il est possible d'enchaîner plusieurs de ces lignes de commande, chacune débutant par la touche de tabulation.

55 Le nom *clean* choisi pour cette cible n'est pas du tout imposé mais correspond à une habitude très courante.

56 L'ordre des règles est sans importance, à l'exception de la première qui définit la cible par défaut.

Vous avez dû constater qu'en fin de compte le programme *make* n'a fait qu'exécuter les commandes que vous saisissiez jusqu'alors dans le terminal. Cependant, si vous saisissez à nouveau dans le terminal la commande :

```
$ make ↵
```

cette dernière doit vous dire que votre programme exécutable est à jour et qu'il n'est donc pas nécessaire d'invoquer les commandes permettant de le fabriquer. Il s'agit là d'un aspect fondamental de cet outil : les commandes ne sont exécutées que si la cible est inexistante ou si ses prérequis sont plus à jour qu'elle-même. Le programme *make* compare en effet les dates de modification des fichiers pour prendre ses décisions. Pour vous en convaincre, modifiez le contenu du fichier *prog01.c* en y ajoutant une ligne vide ou un commentaire, sauvegardez-le et relancez le programme *make*. Vous devriez constater que seule celui-ci est recompilé, puis qu'immédiatement a lieu l'édition de liens, mais que les fichiers *module_A.c* et *module_B.c* ne sont pas recompilés. C'est là que réside l'intérêt principal de cet outil : ***le programme make ne recompile que le strict nécessaire et non l'intégralité du projet***. Ceci permet de gagner un temps précieux lorsqu'on travaille sur de gros projets (notamment avec le langage C++ dont la durée de compilation est bien plus longue).

Reprenez maintenant la démarche consistant à provoquer une modification mineure du code source, mais cette fois-ci sur un fichier d'en-tête, *module_B.h* par exemple. Utilisez à nouveau la commande *make* et observez son comportement : votre modification n'a provoqué aucune nouvelle compilation ce qui est incorrect puisque, comme vu en 2.3.1, la modification d'une déclaration peut avoir des conséquences sévères sur le déroulement du programme. En effet, la définition d'une unité de compilation que nous donnions en 2.3.4 montre que le contenu des multiples fichiers d'en-tête inclus depuis un fichier d'extension *.c* fait partie intégrante des lignes de code que le compilateur analyse pour produire un fichier objet ; la modification de la moindre de ces lignes doit donc provoquer la génération d'un nouveau fichier objet.

Il est vrai que, pour l'instant, le fichier *makefile* ne mentionne aucun fichier d'en-tête ; complétez alors les trois règles qui concernent la phase de compilation afin d'ajouter aux prérequis les fichiers d'en-tête que chaque fichier d'extension *.c* inclut directement ou indirectement (à l'exception de fichiers d'en-tête standards qui, eux, ne devraient pas avoir à être modifiés).

```
prog01.o : prog01.c module_A.h module_B.h
<~TAB~> gcc prog01.c -c

module_A.o : module_A.c module_A.h
<~TAB~> gcc module_A.c -c

module_B.o : module_B.c module_B.h module_A.h
<~TAB~> gcc module_B.c -c
```

Reprenez l'expérimentation pour constater que, dorénavant, la modification de *module_B.h* provoque bien une nouvelle compilation de *prog01.c* et *module_B.c*, et donc une nouvelle édition de liens. Remarquez que *module_A.o* n'a pas besoin d'être reconstruit puisque le fichier d'en-tête modifié ne le concerne pas.

Notre démarche systématique en matière de programmation modulaire et le fichier *makefile* ainsi rédigé assurent désormais une fabrication correcte de notre application en toutes circonstances, sans pour autant nécessiter une compilation systématique de l'ensemble des fichiers de code source.

2.4.2. Généralisation du fichier makefile

Les éléments essentiels à comprendre à propos de l'utilité de l'outil *make* et de la démarche pour rédiger le fichier *makefile* ont été pleinement exposés dans le paragraphe précédent ; c'est ce qu'il vous faudra retenir en priorité.

Il ne s'agit maintenant que d'exposer brièvement quelques facilités pour rédiger un fichier *makefile* de façon à ce qu'il soit facilement modifiable et réutilisable dans des contextes

variés. Les prochains sujets d'exercices s'appuieront en effet sur l'usage d'un fichier *makefile* générique, tel que celui qui est indiqué à la fin de ce sujet.

Un tout premier moyen de donner de la souplesse dans l'usage d'un fichier *makefile* consiste à utiliser des variables. En effet, si nous décidions de changer de version de compilateur, ou bien de changer les options de compilation (ce fut le cas en 2.3.2 pour utiliser le débogueur), il nous faudrait, en l'état actuel, intervenir à de multiples reprises. Une formulation telle que celle-ci (à répéter pour chaque règle de compilation) :

```
CC=gcc
CFLAGS=-g

prog01.o : prog01.c module_A.h module_B.h
<~TAB~> ${CC} prog01.c -c ${CFLAGS}
```

permet au contraire de modifier l'ensemble des commandes de compilation en intervenant seulement sur la variables *CFLAGS*. Remarquez que la valeur d'une telle variable s'obtient avec les symboles *$* (dollar) et *{}* (accolades, ou encore *()*, parenthèses). Le nom de ces variables est libre mais l'usage courant retient souvent les mêmes noms.

(https://www.gnu.org/software/make/manual/html_node/Implicit-Variables.html)

Un autre point qui mérite d'être amélioré concerne le fait que nous devons rappeler à plusieurs reprises les mêmes noms de fichiers dans les règles et les commandes qui les accompagnent, ce qui, au-delà de l'inconfort, est source d'erreurs de saisie. Il existe toutefois des variables fournies automatiquement par le programme *make* pour nous éviter les duplications de noms de fichiers. Voici un nouvel exemple de formulation pour l'édition de liens et la compilation :

```
prog01 : prog01.o module_A.o module_B.o
<~TAB~> ${CC} $^ -o $@

prog01.o : prog01.c module_A.h module_B.h
<~TAB~> ${CC} $< -c ${CFLAGS}

module_A.o : module_A.c module_A.h
<~TAB~> ${CC} $< -c ${CFLAGS}

module_B.o : module_B.c module_B.h module_A.h
<~TAB~> ${CC} $< -c ${CFLAGS}
```

dans lequel les symboles *$@* (dollar-arobas), *$^* (dollar-accent-circonflexe), et *$<* (dollar-inférieur) représentent respectivement la cible (*prog01*), les prérequis (les trois fichiers objets) et le premier des prérequis (*prog01.c*, *module_A.c* ou *module_B.c* selon le cas).

(https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html)

Dans ces conditions, il est même envisageable d'écrire une fois pour toutes des règles génériques précisant les commandes de compilation et d'édition de liens, pour n'avoir qu'à formuler les cibles et les prérequis spécifiques à notre projet (sans répéter la commande) :

```
prog01 : prog01.o module_A.o module_B.o
prog01.o : prog01.c module_A.h module_B.h
module_A.o : module_A.c module_A.h
module_B.o : module_B.c module_B.h module_A.h

% : %.o
<~TAB~> ${CC} $^ -o $@

%.o : %.c
<~TAB~> ${CC} $< -c ${CFLAGS}
```

Ici la notation `% : %.o` s'interprète comme : fabriquer une cible à partir d'un fichier objet ayant le même nom à l'extension près ; il s'agit bien de la commande d'édition de liens. De la même façon, la notation `%.o : %.c` s'interprète comme : fabriquer un fichier objet à partir du code source ayant le même nom à l'extension près ; il s'agit bien de la commande de compilation. À l'utilisation de `make`, les symboles `$^`, `$@` et `$<` prennent bien les valeurs suggérées par les cibles et les prérequis que nous avons spécifiées plus haut pour notre projet.

Un point important qui mérite d'être discuté ici concerne l'énumération des fichiers d'en-tête à prendre en compte dans les prérequis d'une règle ; il s'agit d'un travail fastidieux et source d'erreurs. En effet, il faut non seulement considérer les inclusions directes mais également les inclusions induites par les précédentes. Si nous en oublions, la reconstruction du projet risque d'être incomplète et de donner un programme exécutable incohérent. Si au contraire nous choisissons d'énumérer tous les fichiers d'en-tête du projet pour chaque règle de compilation, alors la modification d'un seul d'entre-eux provoquera la compilation de l'ensemble du projet. Le mieux placé pour faire ce travail de recensement est finalement le préprocesseur lui même !

Ajoutez alors l'option `-MMD` à la variable `CFLAGS` et faites du nettoyage avec `make clean` pour que la commande `make` compile bien tous les fichiers d'extension `.c` avec cette nouvelle option. Dans votre répertoire de travail a dû apparaître pour chacun d'eux un nouveau fichier de même nom mais avec l'extension `.d`. Visualisez le contenu de ces trois nouveaux fichiers : chacun contient une règle qui a exactement la même cible et les mêmes prérequis que ce que nous avions explicitement renseigné dans le fichier `makefile`, si ce n'est que ces informations ont été générées automatiquement par le préprocesseur lorsqu'il a analysé les fichiers en question. Retirez donc ces trois règles du fichier `makefile` et ajoutez à la place cette ligne :

```
-include *.d
```

qui a pour effet d'inclure, s'ils existent, les contenus des fichiers d'extension `.d`, et donc de fournir à nouveau les règles que nous venons de retirer. Bien entendu, la règle de nettoyage doit maintenant supprimer ces fichiers qui sont eux-aussi générés automatiquement :

```
clean :
<~TAB~> rm -f prog01 *.o *.d
```

Dans ces nouvelles conditions, le programme `make` doit toujours se comporter comme précédemment, en ne reconstruisant que ce qui est nécessaire mais sans rien oublier. Nous avons cependant retiré du fichier `makefile` beaucoup d'informations redondantes ou fastidieuses à déterminer.

Finalement, pour être en accord avec la contrainte que nous nous imposerons dans l'usage d'un fichier `makefile` générique tel que celui qui est fourni à la fin de ce sujet, il nous reste à décrire les options de compilation suivantes :

```
CFLAGS=-std=c99 -pedantic -Wall -Wextra -Wc++-compat -Wwrite-strings -Wconversion -MMD -g
```

Au delà des options `-MMD` et `-g` déjà présentées, il s'agit de demander au compilateur d'être le plus sévère possible dans le signalement des avertissements (options `-W...`). Cela augmente nos chances de détecter des maladresses ou des ambiguïtés dans la programmation et améliore la portabilité du code. Nous demandons de plus au compilateur d'être très scrupuleux dans le respect de la norme C99 du langage C ; il s'agit en effet à l'heure actuelle de la version la plus largement utilisée de ce langage.

(http://en.cppreference.com/w/c/language/history)

2.5. Résumé

Ici s'achève cette première séance pratique dans laquelle nous nous sommes familiarisés avec l'environnement de développement et les bonnes pratiques d'organisation du code sans entrer dans les détails du langage C. Ceci pose en effet des bases raisonnables pour envisager des développements dans des conditions réalistes d'usage du langage C pour un ingénieur. Par la suite, tous les exercices pratiques reposeront sur l'organisation modulaire étudiée ici ; dans les

cas les plus simples il s'agira d'un unique module, accompagné de son fichier d'en-tête et d'un programme principal pour en tester les fonctionnalités.

Voici un récapitulatif des notions qu'il vous faudra maîtriser :

- reconnaître et rédiger le prototype d'une fonction,
- distinguer et produire l'appel, la déclaration et la définition d'une fonction,
- distinguer et invoquer le préprocesseur, la compilation et l'édition de liens,
- exécuter le programme produit, y compris dans un débogueur,
- rédiger et utiliser un fichier `makefile` simple (spécifique) et comprendre l'essentiel d'un `makefile` plus élaboré (générique).

et les recommandations que vous devrez désormais toujours suivre en matière de programmation modulaire en langage C :

- déclarer les fonctionnalités (types et prototypes) dans des fichiers d'en-tête,
- inclure ces fichiers d'en-tête dans les modules qui font appel aux fonctionnalités qu'il décrivent,
- inclure chacun de ces fichiers d'en-tête au tout début du module qui en définit le contenu (même si ceci semble parfois superflu, cela assure la cohérence),
- encadrer ces fichiers d'en-tête par la construction `#ifndef`/`#define`/`#endif` en choisissant un nom de *macro* spécifique au fichier en question (cela évite les déclarations redondantes en cas d'inclusions multiples),
- décrire le procédé de fabrication de votre application par un fichier `makefile`,
- prendre en compte dans le fichier `makefile` les fichiers d'en-tête comme dépendance des fichiers `.c` qui les incluent (afin d'en assurer la compilation en cas de modification ; une option du compilateur peut réaliser cette tâche fastidieuse).

Pour vous entraîner, et en vue de passer l'épreuve pratique, veuillez faire un programme modulaire très simple (un module et un programme principal qui invoque ses fonctionnalités par exemple) qui applique systématiquement les consignes récapitulées ici et qui incorpore des éléments du cours sur la découverte du langage C (chapitre 1).

Voici le contenu des fichiers réalisés dans ce sujet, tels qu'ils doivent être si vous avez suivi scrupuleusement les recommandations qui vous ont été données à chaque étape.

```
//---- prog01.c ----
#include <stdio.h> // printf()
#include "module_A.h" // indent()
#include "module_B.h" // doSomething()

int
main(void)
{
indent("* ",1); printf("entering main()\n");
doSomething("| ",2);
indent("* ",1); printf("leaving main()\n");
return 0;
}
```

```
//---- module_A.c ----
#include "module_A.h" // this module
#include <stdio.h> // printf()

void
indent(Tag tag,
       int depth)
{
if(depth!=0)
  { printf(tag); indent(tag,depth-1); }
}
```

```
//---- module_B.c ----
#include "module_B.h" // this module
#include "module_A.h" // Tag, indent()
#include <stdio.h> // printf()

void
doSomething(Tag tag,
            int depth)
{
indent(tag,depth); printf("entering doSomething()\n");
indent(tag,depth); printf("...working hard...\n");
indent(tag,depth); printf("leaving doSomething()\n");
}
```

```
//---- module_A.h ----
#ifndef MODULE_A_H
#define MODULE_A_H 1

typedef const char *Tag;

void
indent(Tag tag,
       int depth);

#endif // MODULE_A_H
```

```
//---- module_B.h ----
#ifndef MODULE_B_H
#define MODULE_B_H 1

#include "module_A.h" // Tag

void
doSomething(Tag tag,
            int depth);

#endif // MODULE_B_H
```

```
#---- makefile (specific version) ----

prog01 : prog01.o module_A.o module_B.o
	gcc prog01.o module_A.o module_B.o -o prog01

prog01.o : prog01.c module_A.h module_B.h
	gcc prog01.c -c

module_A.o : module_A.c module_A.h
	gcc module_A.c -c

module_B.o : module_B.c module_A.h module_B.h
	gcc module_B.c -c

clean :
	rm -f prog01 *.o
```

```
#---- makefile (slightly genericised version) ----

CC=gcc
CFLAGS=-std=c99 -pedantic -Wall -Wextra -Wc++-compat -Wwrite-strings -Wconversion -MMD -g

prog01 : prog01.o module_A.o module_B.o

-include *.d

% : %.o
	${CC} $^ -o $@

%.o : %.c
	${CC} $< -c ${CFLAGS}

clean :
	rm -f prog01 *.o *.d
```

Pour les prochains sujets, nous utiliserons un fichier *`GNUmakefile`* bien plus générique que le précédent. Il produit des programmes exécutables dont le nom commence par le préfixe *`prog`* à partir des fichiers d'extension *`.c`* ayant le nom correspondant. Les autres fichiers sources sont considérés comme des modules utilisés par ces programmes principaux. Il autorise le choix entre la production de code exécutable optimisé ou au contraire instrumenté pour le débogage. Il vise à être utilisé sur plusieurs systèmes d'exploitation ; l'usage de variables autorise des ajustements aux spécificités de la plateforme.

(http://www.enib.fr/~harrouet/Data/Courses/GNUmakefile)

3. C02_Pointers : Pointeurs et tableaux

Le cours d'introduction (chapitre 1) insistait sur l'importance de la notion de type en langage C et en présentait quelques-uns d'usage très courant. Un exercice pratique (chapitre 4) est également consacré à l'étude de la variété de types qui leur sont semblables. Toutefois, il existe en langage C une toute autre catégorie de types qui est extrêmement emblématique de ce langage[57] : il s'agit des pointeurs. Nous verrons que cette notion permet d'interagir finement avec le placement des données en mémoire et, par extension, d'aborder la notion de tableau : une séquence de données désignées par une unique variable.

3.1. Les passages de paramètres

Il est important de retenir qu'en langage C, lors d'un appel de fonction, tous les paramètres, ainsi que la valeur renvoyée, sont transmis par la recopie d'une valeur.

3.1.1. Passage par valeur

Un ***paramètre effectif*** (transmis à l'appel, vocabulaire) est une expression dont l'évaluation donne une valeur qui sert à initialiser une nouvelle variable qui matérialise le ***paramètre formel*** (qui est utilisé à l'intérieur d'une fonction, vocabulaire) correspondant. Une fois initialisé, ce paramètre est totalement indépendant de l'expression qui a fourni sa valeur au moment de l'appel ; la modification de ce paramètre n'a donc aucune influence en dehors de la fonction appelée. Il s'agit d'un ***passage par valeur*** (vocabulaire).

De la même façon, lors de l'utilisation du mot-clef `return`, l'expression qui l'accompagne est évaluée et son résultat est transmis au contexte de la fonction appelante. Une fois cette valeur reçue dans la ***contexte appelant*** (vocabulaire) elle n'est plus du tout liée au contexte de la fonction appelée (qui n'existe d'ailleurs plus, puisque la fonction est terminée).

En voici une illustration sur un exemple :

```
double // nouvelle position x après intégration de la vitesse v pendant dt
integrate(double x, double v, double dt)
{
x+=v*dt;             // le x du contexte local est modifié, pas x0 du contexte appelant
if(x<0.0) { x=0.0; }
return x;            // une copie de la valeur de x est renvoyée vers le contexte appelant
}

// ... dans une autre fonction ...
double x0=... , v0=... , dt=... ; // trois valeurs quelconques
double x1=integrate(x0,v0,dt);    // x0 conserve sa valeur, x1 obtient la nouvelle position
```

Ce mode de passage est généralement adapté aux besoins courants en matière de développement informatique puisqu'il assure une bonne indépendance entre les fonctions, ce qui permet de les composer sans contraintes excessives quant à leur interopérabilité.

57 C'est probablement ce qui constitue l'intérêt essentiel du langage C ; c'est aussi ce qui cristallise l'effroi des débutants vis-à-vis de ce langage !

3.1.2. Passage par adresse

Imaginons maintenant que la fonction `integrate()` de l'exemple précédent prenne en compte une accélération dans l'intégration du mouvement ; il ne s'agit que d'un paramètre supplémentaire à passer par valeur. Cependant, ce nouveau calcul implique désormais la modification de la position et de la vitesse. Il n'est toutefois pas possible de faire renvoyer à une fonction plusieurs valeurs[58].

Le recours à un ***pointeur*** (vocabulaire) permet de simuler un ***passage par référence*** (vocabulaire) c'est à dire un moyen de se référer à une variable qui existe à l'extérieur d'une fonction afin que cette dernière la modifie. L'usage en est fait sur ce nouvel exemple qui servira de support aux explications :

```
void
integrate(double *inout_x, double *inout_v, // passages par adresse (par pointeur)
          double a, double dt)              // passages par valeur
{
double x=*inout_x, v=*inout_v; // acquisition de ce qui est désigné par les pointeurs
x+=(v+0.5*a*dt)*dt;            // calculs sur des variables locales
v+=a*dt;
if(x<0.0) { x=0.0; }
*inout_x=x; *inout_v=v;        // altération de ce qui est désigné par les pointeurs
}

// ... dans une autre fonction ...
double x0=... , v0=... , a=... , dt=... ; // quatre valeurs quelconques
integrate(&x0,&v0,a,dt);                  // x0 et v0 peuvent être modifiées, a et dt non
```

Les noms des deux premiers paramètres de la fonction sont précédés du symbole `*` (étoile) : il s'agit de pointeurs, et ici en particulier de pointeurs vers des données de type `double`. Un pointeur ne contient donc pas directement la valeur utile au calcul mais désigne un emplacement dans la mémoire où cet valeur est stockée : il s'agit de l'***adresse*** (vocabulaire) de cette donnée. La valeur d'une variable de type `double *` est donc l'adresse d'une donnée de type `double` ; recopier une telle variable revient à recopier l'adresse mais pas la donnée désignée.

(http://en.cppreference.com/w/c/language/pointer)

Par conséquent, lors de l'appel à la fonction `integrate()`, les deux premiers paramètres effectifs ne peuvent pas être de type `double` mais doivent être des adresses où sont stockées des données de type `double`. L'opérateur `&` (et-commercial) sert à prendre l'adresse de ce qui lui est associé ; il s'agit de l'opérateur de ***référencement*** (vocabulaire). Ici, les expressions `&x0` et `&v0` représentent respectivement l'adresse des variables `x0` et `v0` qui sont toutes les deux de type `double` ; chacune de ces deux expressions est bien du même type que celui du paramètre formel qui lui correspond.

Désormais, la fonction dispose au travers de ses paramètres `inout_x` et `inout_v`[59] d'un moyen de désigner les variables `x0` et `y0` dont elle ignore tout puisqu'elles appartiennent au contexte appelant. L'opérateur `*` (étoile) sert à accéder à la donnée désignée par le pointeur qui lui est associé ; il s'agit de l'opérateur de ***déréférencement*** (vocabulaire). Ici, les expressions `*inout_x` et `*inout_v` représentent les données de type `double` désignées par ces deux

58 En Python cela était directement possible, en C++17 ce le sera ; dans d'autre langages, tel Java, d'autres solutions plus lourdes (renvoyer un objet, un tableau) sont nécessaires.

59 Le préfixe `inout_` utilisé ici n'a rien d'obligatoire. Il ne s'agit que d'une règle d'écriture personnelle pour attirer l'attention du lecteur du prototype sur le fait qu'un tel pointeur doit désigner une donnée à la valeur déterminée avant l'appel (paramètre d'entrée) et que l'on retrouvera modifiée après (paramètre de sortie).

pointeurs ; dans le cas de l'appel de fonction utilisé dans cet exemple il s'agit respectivement des variables *x0* et *v0* puisque les deux pointeurs les désignent.

(http://en.cppreference.com/w/c/language/operator_member_access)

Remarquez qu'à l'entrée dans la fonction *integrate()* le déréférencement des deux pointeurs nous sert à consulter les données qu'ils désignent. Une copie des valeurs obtenues est utilisée pour initialiser des variables locales qui nous servent à exprimer le calcul. À la sortie de la fonction, la même opération de déréférencement nous permet cette fois-ci de modifier les données désignées par les pointeurs afin de fournir au contexte appelant les résultats du calcul. C'est justement là que réside tout l'intérêt d'avoir eu recours à ces pointeurs.

Puisque le déréférencement d'un pointeur peut faire office de membre gauche d'une opération d'affectation, tout comme une variable, il s'agit également d'une ***lvalue*** (vocabulaire).

Le mode de passage utilisé pour les deux premiers paramètres de cette fonction est qualifié de ***passage par adresse*** (vocabulaire) ou passage par pointeur. Nous avons choisi dans cet exemple de passer les deux variables à modifier de cette façon afin de présenter une écriture homogène. Il aurait toutefois été possible de choisir un passage par valeur, complété par un retour de la fonction, pour un de ces deux paramètres et de conserver le passage par adresse pour l'autre.

Remarquez qu'un pointeur est un type de donnée comme un autre et que du point de vue du langage C un paramètre de type pointeur est passé par valeur comme n'importe quel autre type. La valeur en question représente une adresse dont nous obtenons une copie dans le paramètre formel et c'est le programmeur qui simule un passage par référence en déréférençant ce pointeur.

3.2. Les variables de type pointeur

Nous avons vu qu'un paramètre de fonction pouvait être un pointeur, mais puisqu'il s'agit d'un type à part entière une variable peut également avoir ce type.

3.2.1. Déclaration de pointeurs

Nous reconnaissons la forme d'un tel type au fait qu'un symbole * (étoile) précède le nom de la variable dans sa déclaration. À part ceci, la déclaration de variables de ce type suit les règles habituelles : une seule ou plusieurs déclarations, avec ou sans initialisation...

```
int *v1;                          // un pointeur sur entier de valeur indéterminée
double v2, *v3;                   // un réel et un pointeur sur réel de valeurs indéterminées
int *v4=(int *)0, v5=8, *v6=&v5;  // un entier et deux pointeurs sur entier initialisés
int *v7=v6, *v8, *v9=NULL;        // deux pointeurs initialisées, un autre indéterminé
```

Remarquez que l'étoile est associée au nom de la variable et non à celui du type. Ainsi, sur cet exemple *v2* est de type *double* alors que *v3* est un pointeur sur *double*. Cette notation se comprend simplement comme : **v3* est de type *double*. En effet, en déréférençant *v3* nous avons l'intention de manipuler un réel. De la même façon, *v5* est un entier et non un pointeur.

Comme toute variable de classe de stockage automatique, un pointeur qui n'est pas initialisé a une valeur indéterminée. Dans ce cas cela signifie qu'il désigne une hypothétique donnée située n'importe où dans la mémoire de la machine ! Non seulement le déréférencement d'un tel pointeur non initialisé n'a pas de sens mais il risque au mieux de provoquer un plantage immédiat du programme et au pire de conduire à des modifications arbitraires de la mémoire[60]. ***Dans la pratique il est indispensable d'initialiser un pointeur dès sa déclaration***.

L'initialisation d'un pointeur peut se faire en prenant l'adresse d'une variable du type pointé (c'est le cas de *v6* vis-à-vis de *v5*) ou la valeur d'un autre pointeur du même type (c'est le cas de *v7* vis à vis de *v6*). Toutefois, il est également très courant d'initialiser un pointeur en

60 Ce comportement est bien plus problématique qu'un plantage franc car les conséquences de cette maladresse ne se manifestent que bien après qu'elle ait été commise ; le débogueur ne donne pas d'indices évidents dans ce cas.

indiquant qu'il ne pointe pour l'instant sur aucune donnée utile (sa valeur sera changée ultérieurement). Il suffit pour cela de lui affecter le ***pointeur nul*** (vocabulaire) en effectuant un *cast* de `0` vers le type du pointeur à initialiser (c'est le cas de `v4`) ; une notation alternative consiste à utiliser la *macro* `NULL` (c'est le cas de `v9`) définie dans de nombreux fichiers d'en-tête standards (notamment `stddef.h`).

(http://en.cppreference.com/w/c/types/NULL)

3.2.2. Utilisation d'un pointeur

Au delà de l'usage consistant à désigner une variable inaccessible depuis le contexte local d'une fonction (voir en 3.1.2), un pointeur peut servir à désigner une variable parmi plusieurs selon un critère dépendant des circonstances rencontrées à l'exécution du programme.

```
double x1=... , x2=... ;      // deux valeurs quelconques        // if(x1<x2)
double *p=x1<x2 ? &x1 : &x2 ; // désigner la plus petite         //   { x1+=1.5; x1*=x1; }
*p+=1.5;                      // modifier la plus petite         // else
*p*=*p;                       // ...                             //   { x2+=1.5; x2*=x2; }
```

Dans cet exemple, nous choisissons parmi les variables `x1` et `x2` celle qui sera modifiée par le calcul. Comme illustré par la reformulation en commentaires, le pointeur nous permet d'éviter la duplication du code réalisant le calcul, ce qui est en général une très mauvaise pratique : c'est une source d'erreur de recopie, notamment lors des modifications. Bien entendu, cet exemple est trivial mais il est aisé d'imaginer les proportions que prendrait la duplication de code s'il fallait choisir parmi des variables plus nombreuses et si le calcul était plus complexe.

Nous retrouvons ici les opérations de référencement (`&`) pour obtenir l'adresse d'une variable ou de l'autre, et les opérations de déréférencement (`*`) pour consulter ou modifier la variable désignée par le pointeur.

3.2.3. Un pointeur comme type de retour

Puisqu'un pointeur est un type de donnée comme un autre, il est envisageable pour une fonction de renvoyer un résultat de ce type. Ceci permet par exemple de généraliser l'exemple précédent :

```
double *
chooseBest(double *p1, double *p2, double *p3, double *p4)
{
if(*p1<*p2) { return *p1<*p3 ? (*p1<*p4 ? p1 : p4) : (*p3<*p4 ? p3 : p4); }
else        { return *p2<*p3 ? (*p2<*p4 ? p2 : p4) : (*p3<*p4 ? p3 : p4); }
}

// ... dans une autre fonction ...
double x1=... , x2=... , x3=... , x4=... ; // quatre valeurs quelconques
double *p=chooseBest(&x1,&x2,&x3,&x4);     // désigner la variable correspondant au critère
*p+=1.5;                                   // modifier la variable choisie
*p*=*p;                                    // ...
```

Ici notre critère de choix est plus élaboré et est isolé dans une fonction mais la suite du calcul n'est exprimée qu'en un seul exemplaire et dépend du résultat de cette fonction. Remarquez qu'à une fin de révision, nous avons mélangé l'usage de l'instruction `if-else` et de l'opérateur ternaire pour exprimer les alternatives de notre critère de choix ; nous aurions tout aussi bien pu n'utiliser que l'une ou que l'autre.

Dans l'exemple précédent, le renvoi d'un pointeur par la fonction est tout à fait correct. En effet, les variables `x1`, `x2`, `x3` et `x4` existent dans le contexte appelant la fonction `chooseBest()` : cette fonction peut donc les désigner et les manipuler avec des pointeurs sans craindre leur disparition. Le retour d'un de ces quatre pointeurs vers le contexte appelant désigne toujours une variable existante dans ce même contexte. Cette dernière précision est extrêmement importante, et le contre-exemple suivant illustre une maladresse qu'il ne faut surtout pas commettre :

```
int *
definitelyIncorrectFunction(int i) // !!! NE JAMAIS ÉCRIRE UN FONCTION COMME CELLE-CI !!!
{
int twice=i+i;
return &twice; // l'adresse renvoyée désigne une variable qui va immédiatement disparaître !
}

// ... dans une autre fonction ...
int *v1=definitelyIncorrectFunction(7); // l'adresse obtenue peut désigner n'importe quoi !
int v2=*v1;                             // la valeur obtenue est indéterminée !
```

L'erreur grossière[61] vient ici du fait que la fonction renvoie l'adresse d'une de ses variables automatiques (le problème serait le même en renvoyant l'adresse d'un de ses paramètres qui sont aussi des variables automatiques). En effet, celles-ci sont automatiquement créées à l'entrée dans la fonction (ou le bloc de code), et détruites à sa sortie (voir en 1.3). Le renvoi de l'adresse d'une telle variable est directement lié à la sortie de la fonction donc à la destruction de cette même variable. Après la sortie de la fonction, l'espace mémoire utilisé par la variable désignée par le pointeur renvoyé peut avoir été réutilisé pour stocker une toute autre information. Si le contexte appelant utilise ce pointeur il fait référence à un contenu complètement indéterminé[62].

Par conséquent, nous retiendrons qu'***une fonction ne doit jamais renvoyer l'adresse d'une de ses variables de classe de stockage automatique***. Il est toutefois correct de renvoyer l'adresse d'une de ses variables de classe de stockage statique puisqu'elle existera à la même adresse pendant toute la durée du programme ; toutefois, nous rappelons que l'usage de telles variables est en général peu recommandé.

3.3. Les variables de type tableau

Dans quelques exemples précédents, nous utilisions plusieurs variables de même type auxquelles nous donnions des significations équivalentes afin de désigner l'une ou l'autre d'entre elles par un pointeur. Les tableaux sont justement adaptés à ce propos.

(http://en.cppreference.com/w/c/language/array)

3.3.1. Déclaration et utilisation d'un tableau

Un ***tableau*** (vocabulaire) représente une séquence de données ayant toutes le même type et qui sont désignées par une unique variable ; un indice permet de les distinguer.

```
double data[5];           // data est un tableau de cinq réels dont la valeur est indéterminée
for(int i=0;i<5;++i)      // les élément sont numérotés à partir de zéro
  { data[i]=3.5*i+0.5; }  // désormais chacun des cinq réels a une valeur
for(int i=1;i<5;++i)      // nous nous interdisons d'accéder à un élément inexistant
  { data[i]+=data[i-1]; } // consultation et modification des réels du tableau
printf("%g\n",data[4]);   // 37.5
```

La déclaration d'un tel tableau se reconnaît aux symboles `[ ]` (crochets) qui accompagnent son nom ; ils contiennent une constante littérale entière qui indique le nombre d'éléments qu'il contient[63]. À part les crochets, une telle déclaration est semblable à toute autre déclaration de variable. Si cette déclaration est locale à un bloc de code, les éléments contenus dans ce tableau sont stockés parmi les variables locales de classe de stockage automatique et comme toute autre variable de cette classe de stockage le contenu est initialement indéterminé.

---

61 Il est fort probable que dans un cas évident comme ici le compilateur produise un message d'avertissement.

62 En pratique il peut néanmoins arriver que la valeur attendue soit toujours en place. Il s'agit juste d'une circonstance heureuse pour laquelle aucune garantie n'est donnée. Si cela semble correct une fois, ce ne le sera pas toujours !

63 Depuis la version C99, une expression variable (dont la valeur ne sera connue qu'à l'exécution) peut servir à dimensionner un tableau. Toutefois, cela n'est pas supporté sur tous les compilateurs C et n'est pas autorisé en C++ (même dans sa version moderne). Pour une meilleure portabilité du code nous n'utiliserons pas cette possibilité.

Sur cet exemple, nous initialisons le contenu du tableau dans une boucle de calcul ultérieure qui passe en revue chaque élément. Chacun d'eux est désigné en accolant les symboles `[ ]` (crochets) au nom de la variable pour spécifier un ***indice*** (vocabulaire). Cet indice doit être un entier au sens large : de `char` à `long long` et selon les déclinaisons `signed` ou `unsigned` (voir l'exercice pratique du chapitre 4). Puisque `data` est un tableau de cinq éléments de type `double`, `data[i]` est de type `double` et peut donc recevoir une valeur de ce type. Par convention, ***la numérotation des indices en langage C commence à zéro*** ; le premier élément de ce tableau est donc `data[0]` et le dernier `data[4]` (le nombre d'éléments moins un). Remarquez que cela coïncide parfaitement avec la forme de la très classique boucle `for` dont le compteur commence à zéro et s'arrête juste avant d'atteindre le nombre d'itérations à effectuer.

Dans la seconde boucle de cet exemple nous constatons que la même opération d'***indexation*** (accès selon un indice, vocabulaire) permet aussi bien de consulter un élément du tableau que de le modifier. Un élément d'un tableau peut être utilisé comme une ***lvalue*** (vocabulaire), c'est à dire être placé à gauche d'une opération d'affectation. Remarquez que dans cette seconde boucle, le compteur commence à `1` ; ceci est dû au fait que le corps de la boucle utilise à la fois l'indice courant et le précédent. ***Il est totalement incorrect d'accéder à un élément qui ne soit pas dans les limites d'un tableau***. Si nous le faisions, ceci produirait un comportement indéterminé du programme[64].

3.3.2. Initialisation d'un tableau

Il est possible de spécifier les valeurs des éléments d'un tableau dès sa déclaration ; c'est même recommandé lorsque cela est possible. Cette opération nécessite l'usage d'une nouvelle notation reposant sur les symboles `{ }` (accolades) et `,` (virgule).

(http://en.cppreference.com/w/c/language/array_initialization)

```
int v1[5];                // cinq valeurs indéterminées
int v2[5]={2,3,5,7,11};   // cinq valeurs explicitement spécifiées
int v3[3]={i,4*i+3,f(i)}; // trois valeurs calculées à l'exécution
int v4[]={10,20,30,40};   // les quatre éléments dimensionnent et initialisent le tableaux
int v5[20]={2,3,5,7,11};  // cinq valeurs spécifiées, les quinze autres à zéro
int v6[200]={0};          // deux cents valeurs à zéro
```

Cette notation représente une ***liste d'initialisation*** (vocabulaire) et se comprend ainsi : la paire d'accolades énumère les valeurs des éléments du tableau, dans l'ordre à partir du début, en les séparant deux à deux par une virgule. La variable `v3` nous montre que les valeurs spécifiées ne sont pas nécessairement des constantes ; il doit simplement s'agir d'expressions ayant un type compatible avec celui des éléments du tableau. Comme illustré avec la variable `v4`, si une liste d'initialisation est fournie à la déclaration, il est possible d'omettre le nombre d'éléments du tableau dans les crochets : il est déduit du nombre d'éléments de la liste d'initialisation. Les variables `v5` et `v6`, quant à elles, illustrent le fait que si la liste d'initialisation est plus courte que le nombre d'éléments du tableau, les éléments manquants sont initialisés avec la valeur nulle du type concerné (entier, réel, pointeur...) ; cela permet une économie d'écriture pour les grands tableaux.

Une déclaration de variable (autre qu'un tableau) qui est accompagnée d'une initialisation est en général équivalente à une déclaration sans initialisation (donc avec une valeur indéterminée) immédiatement suivie d'une affectation qui donne sa valeur à cette variable. En ce qui concerne les tableaux, il n'en est rien : ***un tableau ne supporte pas l'opération d'affectation***. La notation qui consiste à lui affecter une liste d'initialisation n'est donc autorisée qu'à sa déclaration et il est impossible d'affecter globalement un tableau à un autre (même s'ils sont du même type et de la même dimension).

64 L'usage d'un élément à l'extérieur des limites d'un tableau est une très fréquente cause d'erreurs en langage C.

```
int v1[3]={10,20,30};                    // correct : trois valeurs explicitement spécifiées
int v2[3];                               // correct : trois valeurs indéterminées
// v2={40,50,60};                        // incorrect : pas d'affectation pour un tableau !
v2[0]=40; v2[1]=50; v2[2]=60;            // correct : affectation des éléments
for(int i=0;i<3;++i) { v2[i]=10*(i+4); } // correct : affectation des éléments
// v2=v1;                                // incorrect : pas d'affectation pour un tableau !
for(int i=0;i<3;++i) { v2[i]=v1[i]; }    // correct : affectation des éléments
```

Le seul moyen d'affecter le contenu d'un tableau consiste à affecter un à un chacun de ses éléments.

3.3.3. Précautions pour l'usage des tableaux et des pointeurs

Malgré leur facilité d'utilisation et une notation indexée très explicite, les tableaux présentent des propriétés qui les rendent assez différents des autres types de données.

La taille d'un tableau

L'exercice pratique du chapitre 4 présente l'opérateur *`sizeof`* qui indique la taille en octets nécessaire pour représenter un type. Cet opérateur s'applique à un nom de type ou à une expression dont le type sera déterminé (une variable par exemple).

(http://en.cppreference.com/w/c/language/sizeof)

Lorsqu'il est appliqué à un tableau, il représente le nombre d'octets nécessaire pour représenter entièrement ce tableau. Cette quantité est équivalente à la multiplication du résultat de l'opérateur *`sizeof`* appliqué à un élément du tableau, par le nombre d'éléments de ce tableau. Une confusion parfois commise consiste à interpréter le résultat de *`sizeof`* comme le nombre d'éléments dans le tableau ; ceci est incorrect, comme illustré sur cet exemple :

```
double data[]={1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9};
printf("size of double: %d\n",(int)sizeof(double));   // 8              (dépend de la plateforme)
printf("size of element: %d\n",(int)sizeof(data[0])); // 8                   (taille d'un double)
printf("size of array: %d\n",(int)sizeof(data));      // 72             (taille de neuf éléments)
int elementCount=(int)(sizeof(data)/sizeof(data[0]));
printf("element count: %d\n",elementCount);           // 9    (nombre d'éléments dans le tableau)
for(int i=0;i<elementCount;++i)
  { printf("%d ",data[i]); }                          // 1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9
```

L'initialisation de la variable *`elementCount`* indique la bonne manière de déterminer le nombre d'éléments d'un tableau si nous souhaitons pouvoir modifier cette quantité à la déclaration sans avoir à corriger les traitements qui exploitent ce tableau. Il est à noter que la taille d'un tableau et celle d'un de ses éléments sont parfaitement connues du compilateur. La division de ces deux quantités l'est donc également ce qui fait que, malgré une écriture plus complexe, cette expression ne prends pas plus de temps à être évaluée à l'exécution que ne le serait une constante littérale.

Limite de la taille d'un tableau

Puisque les tableaux peuvent être déclarés parmi les variables de classe de stockage automatique d'une fonction elles sont créées dans la ***pile d'exécution*** (vocabulaire). Cette zone de la mémoire est de taille variable selon la plateforme d'exécution envisagée. Même si elle peut être redimensionnée sur certains systèmes, elle ne peut pas croître indéfiniment.

Pour cette raison, il faut rester raisonnable quant à la taille des tableaux. Il est courant d'y placer jusqu'à quelques dizaines voire quelques centaines d'éléments. Au delà il est recommandé d'avoir recours à de l'allocation dynamique de mémoire (voir l'exercice pratique du chapitre 6).

Les tableaux multidimensionnels

Dans la documentation qui décrit la déclaration et l'usage des tableaux, il est mentionné que ceux-ci peuvent avoir plusieurs dimensions.

(http://en.cppreference.com/w/c/language/array)

Ceci se matérialise par l'usage de plusieurs paires de crochets (à la déclaration comme à l'indexation) et peut s'interpréter comme un tableau dont chaque élément est un tableau (autant de fois qu'il y a de dimensions).

Dans la pratique, l'usage de tels tableaux est très rarement utile et cette fonctionnalité n'est pas compatible avec l'allocation dynamique de mémoire (voir l'exercice pratique du chapitre 6). De toutes façons, les éléments d'un tableau multidimensionnel sont stockés de manière contiguë en mémoire ce qui constitue finalement un tableau unidimensionnel. L'indexation avec de multiples indices n'est finalement rien d'autre qu'une indexation selon un unique indice prenant en compte les dimensions constantes du tableau.

```
double data1[8]   ={  1.1,2.2,3.3,4.4,   5.5,6.6,7.7,8.8  }; // disposition aplatie
double data2[2][4]={ {1.1,2.2,3.3,4.4}, {5.5,6.6,7.7,8.8} }; // deux lignes, quatre colonnes
int row=1, column=2;
printf("%g %g\n",data1[4*row+column],data2[row][column]);    // 7.7 7.7
```

Nous constatons ici que l'usage de deux paires de crochet revient simplement à dissimuler la multiplication par le nombre d'éléments dans une ligne ; il s'agit d'une constante (`4` ici) et le code exécutable produit est le même dans les deux cas.

Pour toutes ces raisons, nous ne ferons pas usage des tableaux multidimensionnels par la suite et nous utiliserons systématiquement le calcul explicite d'un indice unique afin d'appliquer la même démarche que dans le cas de tableaux obtenus par allocation dynamique de mémoire.

Conversion d'un tableau en pointeur

Comme indiqué plus haut, les tableaux sont assez différents des autres types dans la mesure où ils ne supportent pas l'affectation. Il est toutefois envisageable d'utiliser l'opération d'affectation d'un tableau vers un pointeur. La conversion d'un tableau vers un pointeur sur des éléments de ce tableau est implicite (pas besoin de *cast*) et revient à prendre l'adresse du premier élément de ce tableau. Cependant, ***la conversion d'un pointeur vers un tableau est interdite***.

```
double data[4]={1.1,2.2,3.3,4.4};
double *p=data;      // conversion implicite de double[] vers double *    (équivaut à p=&data[0])
printf("%g\n",*p);   // 1.1              (le pointeur désigne bien le premier élément du tableau)
*p=9.9;              // modification de la donnée désignée par le pointeur
p=&data[2];          // désignation d'un élément au choix dans le tableau
*p=8.8;              // modification de la nouvelle donnée désignée par le pointeur
for(int i=0;i<4;++i)
  { printf("%g ",data[i]); } // 9.9 2.2 8.8 4.4   (les éléments du tableau ont bien été modifiés)
```

Sur cet exemple, nous constatons effectivement que le pointeur `p` pointe bien sur une donnée de type `double` qui est située dans le tableau `data`. Cette situation est, somme toute, très semblable à ce que nous faisions déjà avec des pointeurs sur des variables qui n'étaient pas des tableaux. ***La conversion implicite donnant un pointeur à partir du nom d'un tableau n'est qu'une facilité d'écriture pour prendre l'adresse de son premier élément***.

Puisqu'un pointeur peut désigner un élément d'un tableau, il est raisonnable qu'il puisse aussi désigner les éléments voisins. C'est pour cette raison que les pointeurs supportent également l'opération d'indexation :

```
double data[4]={1.1,2.2,3.3,4.4};
double *p=data;      // conversion implicite de double[] vers double *    (équivaut à p=&data[0])
for(int i=0;i<4;++i)
  { p[i]*=2.0; }     // accès indexé à des éléments désignés par le pointeur
for(int i=0;i<4;++i)
  { printf("%g ",data[i]); } // 2.2 4.4 6.6 8.8   (les éléments du tableau ont bien été modifiés)
```

Nous constatons ici que le pointeur peut facilement être utilisé de la même façon qu'un tableau : l'opération de déréférencement `*p` est strictement équivalente à l'accès indexé `p[0]`

et permet tout aussi bien de consulter que de modifier la valeur désignée. C'est en cela que beaucoup de programmeurs font l'amalgame entre la notion de pointeur et la notion de tableau.

Il est cependant important de garder à l'esprit que cette dualité apparente entre pointeur et tableau n'est pas bidirectionnelle. La distinction se situe essentiellement au niveau de la possibilité d'effectuer une opération d'affectation.

La valeur d'un pointeur est une adresse désignant un emplacement en mémoire où se trouve une donnée du type choisi. Il est tout à fait correct d'affecter une nouvelle valeur à un pointeur : ceci revient à lui faire mémoriser une nouvelle adresse pour désigner l'emplacement d'une autre donnée de même type dans la mémoire. La valeur du pointeur est bien distincte de la valeur de la donnée sur laquelle il pointe : ***un pointeur est une lvalue tout comme la donnée qu'il désigne***.

Un tableau n'a pas vraiment de valeur en soi. Son nom ne désigne que l'adresse d'un emplacement en mémoire à partir duquel ont été placés de manière contiguë ses éléments. Cette adresse n'est pas mémorisée dans le tableau, il s'agit en quelque sorte d'une constante. De la même façon que nous ne pouvons pas affecter une nouvelle valeur à la constante littérale `5`, nous ne pouvons pas affecter une nouvelle adresse à un tableau : il reste là où il se trouve. Seuls les éléments du tableau peuvent recevoir une nouvelle valeur ; ***un tableau n'est pas une lvalue mais ses éléments le sont***.

Passage d'un tableau en paramètre

Si nous souhaitons réaliser une fonction qui opère sur le contenu d'un tableau, nous pouvons une nouvelle fois avoir recours à un pointeur. L'accès indexé avec ce pointeur depuis le corps de la fonction fera bien référence au contenu du tableau situé dans le contexte appelant. Adaptons un exemple déjà traité à l'usage d'un tableau :

```
int // indice de l'élément choisi
chooseBest(double *p, int count)
{
int best=0;
for(int i=1;i<count;++i)
  { if(p[i]<p[best]) { best=i; } }
return best;
}

// ... dans une autre fonction ...
double data[100]={ ... };                // cent valeurs quelconques
int best=chooseBest(data,100);           // désigner l'élément correspondant au critère
data[best]+=1.5; data[best]*=data[best]; // modifier l'élément choisi
best=chooseBest(&data[20],60);           // même chose sur les élément d'indice 20 à 79
data[best]+=1.5; data[best]*=data[best]; // ...
```

Le passage d'un paramètre n'est rien d'autre que l'initialisation du paramètre formel par le résultat de l'évaluation du paramètre effectif. La valeur initiale du paramètre `p` est donc ici un pointeur implicitement déduit du nom du tableau `data`. Remarquez qu'il est prudent d'accompagner un tel pointeur, désignant un tableau, d'un entier indiquant le nombre d'éléments dans ce tableau. Sans cela, le nombre d'itérations devrait être limité par une constante ce qui rendrait cette fonction totalement inutile pour des tableaux d'une autre taille. L'usage d'un pointeur et d'un entier offre même la possibilité de travailler sur un sous-ensemble d'un tableau (lorsque ceci a du sens bien entendu). Cet exemple a été formulé en utilisant l'opération d'indexation et en renvoyant un indice entier ; il aurait été tout aussi correct d'utiliser l'opération de déréférencement et de renvoyer un pointeur.

Il existe une autre notation pour les paramètres désignant des tableaux ; il s'agit d'utiliser la notation à base de crochets comme pour la déclaration de variables de type tableau. Toutefois, cette écriture est extrêmement trompeuse car elle ne traduit pas du tout ce qui semble évident à la lecture.

```
void f1(double *p1);          //
void f2(double p2[]);         // ces trois prototypes sont strictement équivalents !
void f3(double p3[10]);       //
double data[5];
f1(data); f2(data); f3(data); // ces trois appels sont corrects
```

L'usage des crochets dans la déclaration d'un paramètre n'apporte rien par rapport à l'usage d'une étoile. Il n'est pas extrêmement choquant de considérer équivalents les paramètres *p1* et *p2* de cet exemple dans la mesure où l'information de dimension n'est pas fournie : la seule information légitimement attendue est donc l'adresse du début du tableau.

En revanche, la notation utilisée pour *p3* suppose qu'il y a bien dix éléments dans le tableau, mais rien n'est assuré à ce sujet. Il ne s'agit pas d'une copie d'un tableau de dix éléments (puisque l'affectation des tableaux est interdite) mais bien d'un pointeur vers le début du tableau du contexte appelant. Si la fonction *f3()* modifie le contenu du tableau qu'elle reçoit en paramètre, elle modifiera dans les faits le contenu du tableau du contexte appelant ; cela est loin d'être évident à la lecture du prototype et est une source d'erreur !

Plus étonnant encore : l'opérateur *sizeof* appliqué à *p3* ne renvoie pas la taille de dix réels mais la taille d'un pointeur tout comme *p1* et *p2* (voir en 3.4.1) et l'appel à la fonction *f3()* avec un tableau plus petit que ce qui est annoncé pour le paramètre *p3* est autorisé !

Toutes ces raisons font que, ***pour qu'aucune ambiguïté ne subsiste à la lecture du code, le passage d'un tableau en paramètre doit utiliser un pointeur explicitement déclaré avec une étoile dans le prototype de la fonction***.

3.4. Opérations sur les pointeurs

Maintenant que les principaux aspects des pointeurs ont été présentés, voyons comment certaines opérations usuelles pour d'autres types les concernent.

3.4.1. Taille d'une variable de type pointeur

L'exercice pratique du chapitre 4 introduit l'opérateur *sizeof*. Il permet de connaître la taille en octets qui est nécessaire pour représenter une donnée d'un type choisi. Les pointeurs étant des types comme les autres, cet opérateur s'y applique naturellement.

```
printf("  char: %d, *: %d\n",(int)sizeof(char),  (int)sizeof(char *)  ); //   char: 1, *: 8
printf("   int: %d, *: %d\n",(int)sizeof(int),   (int)sizeof(int *)   ); //    int: 4, *: 8
printf("double: %d, *: %d\n",(int)sizeof(double),(int)sizeof(double *)); // double: 8, *: 8
```

Nous constatons que, bien que les types des données désignées par les pointeurs aient des tailles différentes[65], les pointeurs, quant à eux, ont toujours la même taille sur une plateforme d'exécution donnée. Cette taille doit être suffisante pour désigner une adresse dans la mémoire d'une telle plateforme.

La mémoire d'une machine informatique peut être vue de manière extrêmement simplifiée comme une longue séquence de cases numérotées contenant chacune un octet. Une donnée manipulée par un programme occupe un ou plusieurs octets consécutifs de cette séquence ; l'adresse d'une donnée désigne alors le numéro de la case du premier d'entre-eux[66]. Un pointeur, dont la valeur contient une adresse, peut donc être vu comme un compteur de cases mémoire : connaître le numéro de la case mémoire du premier octet d'une donnée et son nombre d'octets (valeur de *sizeof* pour le type pointé) permet de désigner complètement cette donnée.

Les différentes plateformes d'exécution disposent d'une quantité de mémoire plus ou moins grande. Un pointeur doit alors être suffisamment grand pour désigner n'importe quelle case de la mémoire de la machine concernée. Sur un système dit *32-bit*, un pointeur tient sur quatre octets et l'***espace d'adressage*** (vocabulaire) peut théoriquement recouvrir jusqu'à environ quatre milliards d'octets. Sur un système dit *64-bit*, un pointeur tient sur huit octets et l'espace d'adressage peut théoriquement recouvrir jusqu'à environ dix-huit milliards de milliards

65 Ces tailles varient d'une plateforme d'exécution à une autre ; ceci est discuté au chapitre 4.

66 La notion d'adresse est étudiée en détail dans les matières S5-MIP et S6-MIP de l'ENIB.

d'octets. Bien entendu, la quantité de mémoire effectivement présente sur la machine informatique peut être bien moindre que ce que son espace d'adressage autorise.

3.4.2. Arithmétique des pointeurs

Puisque la valeur d'un pointeur désigne le numéro d'une case mémoire, il n'est pas tout à fait aberrant de l'assimiler à un entier (même si c'est incorrect du point de vue du langage C) et d'y appliquer des opérations arithmétiques pour désigner les cases mémoire voisines.

Le langage C autorise en effet l'addition et la soustraction d'un entier à un pointeur. Cela n'a de sens que lorsque le pointeur désigne une valeur au sein d'un tableau ; lui additionner ou lui soustraire un entier revient à produire une adresse qui désigne une donnée éloignée d'autant d'éléments.

```
double data[10]={0.0,1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9};
double *p=data+3;         // pointeur de début de tableau avancé de trois éléments
printf("%g\n",*p);        // 3.3                              (le pointeur désigne bien data[3])
++p; *p+=40.04;           // passage à l'élément suivant, puis modification de la donnée pointée
p=data+9;                 // désignation du dernier élément du tableau
printf("%g\n",*p);        // 9.9                              (le pointeur désigne bien data[9])
p-=2; *p+=70.07;          // recul de deux éléments, puis modification de la donnée pointée
for(int i=0;i<10;++i)
  { printf("%g ",data[i]); } // 0.0 1.1 2.2 3.3 44.44 5.5 6.6 77.77 8.8 9.9
```

Remarquez que les seules opérations arithmétiques qui s'appliquent aux pointeurs sont celles qui reposent sur l'addition ou la soustraction (la multiplication ou la division n'aurait aucun sens). Si le langage autorise de telles opérations sur n'importe quel pointeur, il va sans dire que cela ne produit l'effet escompté que si le pointeur désigne un élément d'un tableau et si son déplacement reste dans les limites du tableau. Si nous appliquions ceci à un pointeur désignant une simple variable nous tomberions à côté de celle-ci, ce qui produirait un résultat indéterminé donc une erreur dans le programme.

L'arithmétique des pointeurs permet d'assurer le lien entre les opérations d'indexation, de référencement et de déréférencement.

```
double *p= ... ; // adresse d'un élément dans un tableau suffisamment grand
int i= ... ;     // un indice convenable vis-à-vis de p et de son tableau d'origine
if(  p[i] == *(p+i) ) { printf("this is always true!\n"); } // les expressions de part
if(  p[0] == *p     ) { printf("this is always true!\n"); } // et d'autre de ces égalités
if( &p[i] == p+i    ) { printf("this is always true!\n"); } // sont strictement équivalentes
```

Remarquez que finalement l'opération d'indexation est complètement facultative : il ne s'agit que d'un confort d'écriture pour éviter d'avoir recours à l'arithmétique des pointeurs et au référencement dans les situations où ceux-ci risquent d'être difficilement lisibles.

Un entier peut être ajouté à un pointeur mais deux pointeurs ne peuvent pas être additionnés (cela n'aurait aucune signification). En revanche, il est envisageable de soustraire deux pointeurs s'ils désignent des données de même type. Le résultat d'un telle soustraction indique le nombre d'éléments de ce type qui les séparent[67], ce qui est cohérent avec le fait que deux pointeurs identiques sont séparés d'une distance nulle.

```
double data[10]={0.0,1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9};
double *p1=data+2, *p2=data+8;
printf("%d %d\n",(int)(p1-data),(int)(p2-data)); // 2 8
printf("%d %d\n",(int)(p1-p2),  (int)(p2-p1));   // -6 6
```

Bien entendu, les deux pointeurs doivent désigner des données situées dans un même tableau ; dans le cas contraire, la valeur de cette différence n'a aucune signification utile. Remarquez que le type de cette différence n'est pas un entier habituel. Il doit être assez grand pour exprimer le plus grand écart entre adresses qui puisse exister dans la machine, ce qui fait

67 La différence de pointeurs n'indique pas un écart en octets ; c'est une confusion commune chez les débutants !

qu'il doit avoir la même taille qu'un pointeur (très grand sur un système *64-bit*). Cet entier est signé car il peut indiquer un écart positif ou négatif selon la disposition relative des données désignées par les pointeurs. Le nom usuellement donné (un `typedef` fourni par le fichier d'en-tête standard `stddef.h`) à ce type est `ptrdiff_t`. Dans cet exemple nous le convertissons en `int` pour l'affichage car nous savons que l'écart entre les éléments ne sera pas énorme.

Par extension de la différence des pointeurs, et de la même façon que nous évaluons l'égalité et la différence de deux pointeurs de même type avec les opérateurs `==` et `!=`, nous pouvons les comparer avec les relations `<`, `<=`, `>` et `>=`.

Remarquez que, étant donné qu'elle n'a de sens pratique qu'au sein d'un unique tableau, ***l'arithmétique des pointeurs n'est rien d'autre que l'arithmétique des indices d'un tableau***.

Toutes ces propriétés font que les itérations d'une boucle peuvent tout aussi bien être contrôlées par des pointeurs.

```
double                                                  // double
arraySum(double *begin, double *end)                    // arraySum(double *p, int count)
{                                                       // {
double sum=0.0;                                         // double sum=0.0;
for(double *p=begin;p<end;++p)                          // for(int i=0;i<count;++i)
  { sum+=*p; }                                          //   { sum+=p[i]; }
return sum;                                             // return sum;
}                                                       // }

// ... dans une autre fonction ...
double data[100]={ ... }; // cent valeurs quelconques
double sum=arraySum(data,data+100);                     // double sum=arraySum(data,100);
```

Les deux variantes présentées ici effectuent le même traitement ; les habitudes des programmeurs sont partagées entre ces deux types d'écriture. Sur un exemple aussi simple que celui-ci, les deux formulations sont équivalentes en terme de code exécutable généré. Dans le cas d'algorithmes plus complexes, la formulation à base d'indices entiers est toutefois préférable car il expose plus facilement au compilateur les possibilités d'optimisation[68].

3.4.3. Valeur booléenne d'un pointeur

En 3.2.1 nous avons vu qu'un pointeur pouvait se faire attribuer une valeur nulle. Lorsqu'un pointeur nul est déréférencé, ceci provoque un plantage immédiat du programme qui est facile à détecter dans un débogueur. Nous pouvons alors investiguer pour comprendre l'origine de ce pointeur nul et corriger le code en conséquence.

Lorsque dans un programme il est envisageable qu'un pointeur soit tantôt nul, tantôt initialisé pour effectivement désigner une donnée, il nous faut distinguer ces deux cas.

```
void
showPointer(int *ptr)
{
if(ptr) // équivalent à  if(ptr!=(int *)0)  ou encore  if(ptr!=NULL)
  { printf("it points to the value %d\n",*ptr); }
else
  { printf("it points nowhere!"); }
}
```

Comme pour les types numériques, on considère qu'un pointeur a la valeur booléenne fausse s'il est nul et vraie dans tous les autres cas (quelle que soit l'adresse exacte qu'il indique).

68 De multiples pointeurs peuvent laisser supposer des situations d'*aliasing* qui n'ont pas forcément lieu d'être. Cette notion sort du cadre de cette initiation au langage C.

Attention : ***un pointeur non ou mal initialisé n'est pas forcément nul***. Le test booléen sur un tel pointeur indiquera probablement une valeur vraie alors qu'aucune donnée utile n'est exploitable à l'adresse qu'il indique[69].

3.4.4. Conversion du type d'un pointeur

Puisqu'il est autorisé de convertir un type vers un autre, que ce soit implicitement ou explicitement par un *cast*, il est tentant d'envisager la même chose à propos des types qui sont des pointeurs : s'il existe des conversions implicites et explicites entre les types `int` et `double`, pourquoi n'en serait-il pas de même entre les types pointeur sur `int` et pointeur sur `double` ?

La différence essentielle tient dans le fait que les pointeurs ne contiennent pas les valeurs utiles qu'ils désignent ; ils se contentent de les référencer par leur adresse. Une adresse garde la même forme quel que soit le type de la donnée qu'elle désigne : ce n'est qu'un numéro de case mémoire. Le type associé à la déclaration d'un pointeur indique donc comment interpréter les octets occupant cet emplacement. Par conséquent, convertir le type d'un pointeur revient à changer l'interprétation qui sera faite de ces octets mais ça ne les change aucunement. C'est pour cette raison qu'aucune conversion de pointeur n'est implicitement autorisée.

Il est néanmoins possible d'avoir recours au *cast* pour forcer la conversion d'un type de pointeur vers un autre.

```
double d=12.34, *p1=&d;
int *p2=(int *)p1;         // conversion extrêmement risquée (cela n'a probablement aucun sens)
printf("%g %d\n",*p1,*p2); // 12.34 2061584302                    (une valeur entière possible)
```

La valeur entière indiquée dans cet exemple n'a aucune signification utile et est dépendante de l'architecture spécifique de la plateforme d'exécution. Si ce n'est pour quelques raisons pédagogiques, la conversion du type d'un pointeur n'est utile en pratique que dans quelques rares cas identifiés par des experts.

3.4.5. Le qualificatif `const`

Le prétexte qui nous a servi à introduire les pointeurs dans ce chapitre reposait sur la nécessité d'offrir à une fonction le moyen de modifier des variables de son contexte appelant (voir 3.1.2). Plus loin nous avons constaté que le passage d'un tableau en paramètre à une fonction revenait à passer un pointeur sur son premier élément (voir 3.3.3). Il s'ensuit que le passage d'un tableau en paramètre à une fonction offre à cette dernière la possibilité de le modifier. Or, la raison de ce passage pourrait être uniquement motivée par la nécessité de consulter le tableau. En l'état, le contexte qui appelle une telle fonction ne peut savoir, sans connaître les détails de l'algorithme de la fonction, si le tableau utilisé en paramètre effectif sera modifié ou bien simplement consulté.

Pour clarifier cette situation, le mot-clef `const` peut qualifier un paramètre de type pointeur afin d'indiquer le fait qu'il n'autorise que la consultation (pas la modification). Non seulement, à la lecture de ce prototype l'ambiguïté est levée, mais dans la définition de la fonction toute tentative de modification de la donnée pointée donnera lieu à une erreur de compilation.

Il est correct de convertir implicitement un pointeur qui n'est pas qualifié de `const` vers un autre qui le serait. En revanche, la conversion opposé est interdite puisque cela reviendrait à enfreindre la contrainte qui garantit que le donnée désignée ne doit pas être modifiée. ***Il est cependant possible d'avoir recours à un cast pour se débarrasser du qualificatif*** `const` ***mais cette conversion explicite est une extrêmement mauvaise pratique[70] qui est révélatrice d'un défaut de conception majeur dans le programme*** (deux décisions contradictoires).

69 L'usage de pointeurs à la valeur indéterminée ou invalide est une très fréquente cause d'erreurs en langage C.

70 Ceci peut même conduire à un plantage de l'application si le pointeur qualifié de `const` désigne une zone mémoire qui n'est pas modifiable (du point de vue du système d'exploitation).

Voici, à titre d'exemple, une fonction dont le prototype signifie clairement que les données accédées par le pointeur passé en paramètre (un tableau de *count* éléments ici), ne seront pas modifiées :

```
void
showArray(const double *p, int count)
{
for(int i=0;i<count;++i)
  { printf("%g ",p[i]); } // consultation des valeurs pointées
// *p+=10.0;              // incorrect : modification d'une valeur pointée interdite
// int *p2=p;             // incorrect : perte du qualificatif const interdite
}

// ... dans une autre fonction ...
double data[100]={ ... }; // cent valeurs quelconques
const double *p=data+20;  // ajout implicite du qualificatif const autorisé
showArray(p,60);
showArray(data,100);      // ajout implicite du qualificatif const autorisé
```

La position du mot-clef *const* dans la déclaration du pointeur est importante : ici il est proche du type de la donnée pointée (avant l'étoile) ce qui signifie bien que c'est cette donnée qui est constante. Le pointeur lui n'est pas constant, c'est à dire qu'on peut lui affecter une nouvelle adresse, mais il désignera à nouveau une donnée constante.

(http://en.cppreference.com/w/c/language/const)

Le choix de l'usage du mot-clef *const* dans la déclaration des prototypes de fonctions est essentiel et fait partie intégrante de la réflexion pour définir une interface de programmation. En effet, si le prototype d'une fonction n'utilise pas ce qualificatif pour un paramètre de type pointeur alors que sa définition ne modifie pas les données pointées, cette fonction ne sera malgré tout pas utilisable avec comme paramètre effectif un pointeur qui aurait le qualificatif *const*. ***À chaque fois qu'un pointeur ou un tableau doit être transmis en paramètre à une fonction, il faut s'interroger sur le fait que les données désignées puissent être modifiées ou non par cette fonction et utiliser le mot-clef*** *const* ***en conséquence***.

Le qualificatif *const* n'est pas limité aux pointeurs ; il peut s'appliquer à tout type de donnée et il est même conseillé d'y avoir recours pour signifier qu'une donnée ne doit pas changer dans un algorithme. Ceci permet d'éviter une modification involontaire causée par une erreur de programmation et peut même aider le compilateur à optimiser le code exécutable. Dans le cas des pointeurs, un mot-clef *const* peut être placé après l'étoile afin de signifier que c'est le pointeur qui ne peut pas être modifié (il ne peut pas désigner une nouvelle adresse).

3.4.6. Pointeur sur pointeur

Nous avons vu qu'une donnée de type pointeur est une donnée comme une autre ; sa valeur est simplement une adresse. Il est envisageable d'évaluer l'adresse où cette donnée est elle-même stockée et de l'affecter à une autre variable de type pointeur. Cette dernière est donc un pointeur vers un autre pointeur. Ce procédé peut être répété à l'envie.

La notation utilisée pour déclarer le type d'un pointeur consiste à placer un symbole * (étoile) devant le nom de la variable ; il suffit d'ajouter une nouvelle étoile pour dénoter chaque ***indirection*** (vocabulaire) supplémentaire.

```
double data[4]={1.1,2.2,3.3,4.4};
double *p1=data+1;
printf("%g\n",*p1);          // 2.2
double **p2=&p1;             // p2 désigne p1
*p2+=2;                      // modifier *p2 revient à modifier p1
printf("%g %g\n",*p1,**p2);  // 4.4 4.4     (déréférencement de p1, double-déréférencement de p2)
```

Le recours à de multiples indirections imbriquées est généralement peu recommandé en langage C. Une telle structuration arborescente des données est souvent révélatrice d'une montée en abstraction dans le problème et ce n'est pas à ce niveau que le langage C est le plus utile. Le langage C permet une exploitation efficace des capacités de calcul de la machine en contrôlant finement le placement de données. Au contraire, les multiples indirections ont tendance à fragmenter l'usage de la mémoire ce qui allonge considérablement le temps d'accès aux données et compromet les performances.

Pour poursuivre dans l'évocation de pointeurs élaborés, sachez qu'il existe des pointeurs sur des données indéterminées : `void *`. Ils contiennent bien une adresse mais ne précisent pas le type de la donnée désignée, ce qui interdit l'indexation, le déréférencement et l'arithmétique des pointeurs. Ils seront utilisés dans l'exercice pratique du chapitre 6. Les fonctions peuvent également être désignées par des pointeurs ; l'usage d'une telle fonctionnalité sera découvert dans l'exercice pratique du chapitre 11.

3.5. Résumé

Ce cours visait principalement à introduire les pointeurs qui représentent une notion essentielle et très spécifique du langage C. Nous avons vu qu'ils apportaient des solutions à plusieurs besoins pratiques[71] :

- accéder à des données externes au contexte d'une fonction afin de les modifier,
- désigner une donnée parmi plusieurs selon les circonstances applicatives,
- transmettre un tableau de données à une fonction,
- exploiter le contenu d'un tableau.

La syntaxe et l'interprétation de leur déclaration et de leurs opérations (référencement, déréférencement, indexation, arithmétique) doivent être parfaitement maîtrisées pour envisager le moindre usage raisonnable des pointeurs. Il en va de même de la distinction entre les pointeurs et les tableaux dans la dualité imparfaite qui les lie.

71 D'autres cas d'utilisation, mais qui restent cependant proches de ceux-ci, existent ; nous le verrons notamment dans l'exercice pratique du chapitre 6.

4. L02_Types : Variété de types

Motivation :

Comme vu dans le cours du chapitre 1, le langage C impose le choix d'un type pour chacune des données manipulées. Le cours introductif a présenté les quelques-uns d'entre eux qui sont les plus couramment utilisés ; il faut effectivement les choisir en priorité si aucune contrainte n'impose l'usage d'un type plus judicieux. Il existe toutefois une variété de types qui sont semblables à ceux-ci. Ce sont également des entiers et des réels mais ils diffèrent essentiellement par la plage de valeurs qu'ils peuvent recouvrir.

Ce sujet propose de découvrir les principales propriétés de cette variété de types afin de mettre en évidence les limites qu'ils imposent quant au bon fonctionnement d'un programme. Ceci permettra d'acquérir les éléments de décision pour choisir les types les mieux adaptés aux problèmes qui risquent de franchir de telles limites.

Consignes :

Le travail demandé ici est extrêmement guidé puisqu'il s'agit de provoquer des cas problématiques afin de discuter de leur cause et éventuellement d'envisager des solutions. Veilliez donc à suivre scrupuleusement chacune des étapes en vous assurant à chaque fois d'avoir bien observé et compris ce qui vous est décrit.

4.1. Mise en place du programme

Ouvrez un terminal et grâce à la ligne de commande du *shell*, placez vous dans le répertoire de travail choisi pour ce sujet. Saisissez par exemple :

```
$ mkdir S3PRC_L02_Types ↲
$ cd S3PRC_L02_Types ↲
```

Placez dans ce répertoire le fichier `GNUmakefile` générique mis à votre disposition.

(http://www.enib.fr/~harrouet/Data/Courses/GNUmakefile)

Nous suivrons en effet la démarche de programmation modulaire présentée au chapitre 2.

Vous devrez tout d'abord réaliser une fonction `displaySize()` qui n'attend aucun paramètre et ne renvoie aucun résultat. Elle sera déclarée dans le fichier d'en-tête `typeProperties.h` et définie dans le fichier `typeProperties.c`. Le programme principal sera rédigé dans le fichier `prog02.c` et appellera cette fonction. La fonction `displaySize()` se contentera pour l'instant d'afficher son nom. La rédaction de ces trois fichiers doit respecter les consignes usuelles (chapitre 2).

Fabriquez ce programme et exécutez-le afin de vérifier que tout est en place. En l'état, il ne fait qu'afficher le nom de la fonction `displaySize()` et se termine immédiatement :

```
$ make ↲
$ ./prog02 ↲
```

4.2. La taille des types

Un critère très couramment considéré lors du choix d'un type de données concerne la taille qu'il occupe dans la mémoire. Ceci est particulièrement important lorsqu'une très grande quantité de données de ce type doit être manipulée et mémorisée[72].

4.2.1. L'opérateur `sizeof`

Le langage C dispose à ce propos de l'opérateur `sizeof` qui donne la taille d'un type exprimée en *bytes* (c'est à dire en octets dans la pratique).

(http://en.cppreference.com/w/c/language/sizeof)

Il peut s'utiliser de deux façons :

```
taille_du_type_choisi=sizeof(nom_d_un_type);
taille_du_type_de_l_expression=sizeof(une_expression);
```

Si son usage est trivial dans le cas où il est appliqué à un type, il nécessite toutefois un éclaircissement dans le cas d'une expression. Cette dernière est analysée comme à l'accoutumée par le compilateur ce qui conduit à la détermination de son type. Le type déterminé est alors utilisé par l'opérateur `sizeof` ; cependant, l'expression en question n'est pas évaluée ni exécutée ! Si elle exprime des effets de bord, ceux-ci n'auront pas lieu. Dans la pratique cette notation est très souvent utilisée pour se référer au type d'une expression qui se réduit à une simple variable : la taille occupée par cette variable.

Complétez tout d'abord la fonction `displaySize()` pour afficher de manière claire la taille du type `int`. Puisqu'elle exprime un nombre d'octets, cette taille est une grandeur entière ; choisissez le format de l'appel à la fonction `printf()` en conséquence.

Relancez la fabrication du programme ; celle-ci doit vous avertir du fait que cette taille n'a pas tout à fait le type d'un simple entier. En effet, une relecture attentive de la documentation de `sizeof` doit vous apprendre que son type est `size_t`.

(http://en.cppreference.com/w/c/types/size_t)

Il s'agit d'un nom de type (`typedef`), proposé par les fichiers d'en-tête standards (notamment `stddef.h`), désignant un entier non-signé (`unsigned`) capable de représenter la taille d'une donnée quelconque tenant dans la mémoire de la machine[73]. Dans un environnement *64-bit* (c'est le cas pour les machines des salles de labo de l'ENIB) cette taille est potentiellement gigantesque (même si la machine particulière utilisée en pratique dispose d'une quantité de mémoire moindre).

Puisqu'ici nous déterminons la taille d'un simple entier nous pouvons sans risque convertir (*cast*, voir en 1.3.3) le résultat de l'opérateur `sizeof` vers le type `int` afin de faire disparaître le message d'avertissement. Profitez-en alors pour procéder de même en affichant également la taille du type `size_t`.

Fabriquez à nouveau le programme (ce qui ne doit plus provoquer d'avertissement) et exécutez-le afin de constater les tailles respectives des types `int` et `size_t`.

4.2.2. Comparaison de la taille des types

Le résultat précédent illustre le fait que des entiers peuvent avoir des tailles différentes. Consultez donc la variété de types décrite sur cette page de documentation :

(http://en.cppreference.com/w/c/language/arithmetic_types)

Complétez alors la fonction `displaySize()` pour afficher de manière claire la taille des types :

`bool`[74], `char`, `short`[75], `int`, `long`, `long long`, `float`, `double`, `long double`

72 Les tableaux (chapitre 3) et l'allocation dynamique (chapitre 6) sont particulièrement concernés par ce critère.

73 L'allocation dynamique (chapitre 6) permet de manipuler un très grand volume de données.

74 L'inclusion du fichier d'en-tête standard `stdbool.h` est nécessaire pour ce type.

75 Le type `short` est une abréviation pour `short int`, tout comme `long` l'est pour `long int` et `unsigned` pour `unsigned int`.

Fabriquez à nouveau le programme et exécutez-le afin de constater les tailles de tous ces types. Vérifiez en particulier que les valeurs affichées respectent bien ces relations[76] :

`sizeof(char)==1`,
`sizeof(char)<=sizeof(short)<=sizeof(int)<=sizeof(long)<=sizeof(long long)`,
`sizeof(short)>=2`, `sizeof(int)>=2`, `sizeof(long)>=4` et `sizeof(long long)>=8`.

4.2.3. Les entiers de taille déterminée

Au delà des types booléen et réels, l'expérience précédente montre en particulier qu'il existe des types entiers ayant des tailles variées. Seulement, la norme du langage C ne spécifie pas de taille exacte pour ces types (à l'exception de `char`). Dans ces conditions, il est difficile d'écrire du code portable qui réponde à des exigences précises sur la taille des entiers. Heureusement, le fichier d'en-tête `stdint.h` fournit des types (par l'usage de `typedef`) dont le nom assure le respect de la taille annoncée sur la plateforme d'exécution visée.

(http://en.cppreference.com/w/c/types/integer)

Complétez alors la fonction `displaySize()` pour afficher de manière claire la taille des types :

`int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`

Fabriquez à nouveau le programme et exécutez-le afin de vérifier que tous ces types tiennent bien leur promesse en terme de taille.

Il peut être tentant de se précipiter sur un usage exclusif de ces types à taille déterminée, mais les experts s'accordent à dire que le choix par défaut pour un entier doit se porter sur le type `int` si aucune contrainte spécifique n'est exprimée (un simple compteur par exemple)[77]. Le recours à des entiers non-signés (`unsigned`) ou de taille spécifique ne doit être envisagé que dans de très rares cas : opérations *bit-à-bit* (chapitre 9), plages de valeurs strictement définies (interaction avec du matériel)...

4.3. Dépassement de capacité

Puisque tous les entiers ont une certaine taille (quelle qu'elle soit), ils ne peuvent décrire qu'un ensemble fini de valeurs. Nous nous intéressons ici à ce qu'il se passe lorsque nous sortons de cette plage de valeurs.

4.3.1. Conversion vers un type insuffisamment large

Vous devez maintenant réaliser une fonction `detectOverflow()` qui attend un paramètre entier nommé `limit` et ne renvoie aucun résultat. Elle sera déclarée dans le fichier d'en-tête `typeProperties.h` et définie dans le fichier `typeProperties.c`. Il faudra ajouter au programme principal du fichier `prog02.c` un appel à cette fonction avec la valeur `10000` (dix mille). Cette nouvelle fonction doit elle aussi commencer par afficher son nom et la valeur de son paramètre `limit`.

Dans la fonction `detectOverflow()`, réalisez une boucle `for` qui compte avec un `int` de `0` à `limit`. Dans le corps de cette boucle, affectez le compteur à une variable de type `uint16_t` (un *cast* est nécessaire). Utilisez alors une alternative simple (`if`) pour tester si la variable en question est différente de celle du compteur ; dans ce cas, affichez clairement ces deux valeurs et provoquez une terminaison prématurée de la boucle avec l'instruction `break`.

Fabriquez à nouveau le programme et exécutez-le afin d'observer l'effet de cette boucle. En l'état, aucune différence de valeurs ne doit être signalée. Complétez à nouveau le programme principal par un nouvel appel à cette fonction qui utilise `100000` (cent mille) comme paramètre. Après une nouvelle fabrication, l'exécution de ce programme doit maintenant signaler une différence de valeurs dans la boucle de la fonction `detectOverflow()`. Observez alors les valeurs indiquées et confrontez-les à la taille du type `uint16_t` utilisé : vous devez interpréter

76 Dans le cas où un *byte* correspond à un octet (huit *bits*).

77 Le type `int` représente généralement le type entier qui est le plus efficacement exploité sur la plateforme d'exécution courante.

la valeur observée en terme de puissance de deux. Pour confirmer votre intuition, ajoutez après cette boucle l'affichage de la constante entière `UINT16_MAX` et relancez l'expérience.

Vous venez, d'une part, de mettre en évidence la plage de valeurs pouvant être décrites par une variable de type `uint16_t`, et d'autre part, de constater le comportement d'un tel entier non-signé lorsqu'un dépassement de capacité survient : le compte repart à `0`. En effet, les *bits* de poids forts manquants sont ignorés et ne subsistent que les *bits* de poids faibles. De la même façon, attribuer à un entier non-signé une valeur négative provoque un rebouclage vers les grandes valeurs positives.

Ce comportement est parfaitement décrit dans la norme du langage C mais uniquement pour les entiers non-signés. ***Il est donc parfaitement correct d'écrire un algorithme qui compte sur le fait qu'un entier non-signé trop grand ou trop petit reboucle vers l'autre extrémité de la plage de valeurs***.

Intervenez maintenant dans la fonction `detectOverflow()` pour changer le type `uint16_t` en `int16_t` et pour ajouter l'affichage de la valeur des constantes entières `INT16_MIN` et `INT16_MAX` après la boucle. Relancez l'expérience et observez les valeurs reportées lors du dépassement de capacité. De manière assez similaire à l'expérience précédente, vous devez constater que lorsque le compteur dépasse `INT16_MAX`, la valeur obtenue dans la variable de type `int16_t` reboucle vers `INT16_MIN`.

Attention toutefois, même si dans la pratique nous constatons un tel rebouclage, la norme du langage C précise qu'un dépassement de capacité sur un entier signé (que ce soit par les valeurs positives ou négatives) produit un comportement indéfini. ***Il est donc totalement incorrect d'écrire un algorithme qui compte sur le fait qu'un entier signé trop grand ou trop petit reboucle vers l'autre extrémité de la plage de valeurs***. Ce comportement indéterminé offre au compilateur de langage C une très bonne opportunité pour optimiser certains algorithmes. En effet, l'utilisation d'un compteur signé permet parfois de faire l'économie de précautions qui seraient indispensables dans le cas d'un compteur non-signé : un rebouclage éventuel pourrait être légitimement prévu par un algorithme utilisant un entier non-signé alors que cette éventualité peut être sereinement ignorée sur un entier signé[78].

Dans les deux cas traités ici, signé et non-signé, le compilateur sait qu'il y a un risque de perte d'information lors de l'affectation d'un entier vers un autre plus petit, c'est pourquoi il produit un avertissement pour attirer notre attention. Pour lui indiquer que nous assumons ce choix, c'est le cas dans cet exercice, nous devons utiliser l'opération de conversion explicite de type (*cast*). Bien entendu, ce que nous avons constaté ici autour d'entiers de seize *bits* se généralise à toutes les tailles d'entiers. Pour terminer ce volet, prenons alors connaissance des ordres de grandeur de quelques types entiers usuels. Affichez dans la fonction `detectOverflow()` la valeur des constantes entières `INT_MIN`, `INT_MAX`, `CHAR_MIN` et `CHAR_MAX` du fichier d'en-tête standard `limits.h`.

(http://en.cppreference.com/w/c/types/limits)

Observez les valeurs limites affichées par l'exécution de cette nouvelle version du programme et confrontez-les aux tailles reportées par la fonction `displaySize()`.

4.3.2. Incrémentation excessive

Nous réutilisons ici le prétexte du dépassement de capacité pour mettre en œuvre quelques éléments du cours sur les éléments de base du langages (chapitre 1) à une fin d'entraînement.

Vous devez maintenant réaliser une fonction `tributeToChuck()` qui n'attend aucun paramètre et ne renvoie aucun résultat. Elle sera déclarée dans le fichier d'en-tête `typeProperties.h` et définie dans le fichier `typeProperties.c`. Il faudra ajouter au programme principal du fichier `prog02.c` un appel à cette fonction. Cette nouvelle fonction doit elle aussi commencer par afficher son nom.

Inventez (avec `typedef`) un nouveau nom de type `TestInt` qui ne désigne rien d'autre que le type `uint32_t` ; il ne sera utilisé que dans cette fonction et doit donc lui être local. Réalisez

78 Ceci est particulièrement important dans le contexte de la vectorisation automatique de calculs (qui dépasse largement le cadre de cette découverte du langage C).

alors une boucle `for` utilisant un compteur de type `TestInt` qui commence à `0` et qui est incrémenté à chaque itération ; cependant, cette boucle n'a pas de condition, elle se poursuit donc infiniment. Dans le corps de cette boucle, il vous faudra détecter si la valeur courante de ce compteur est inférieure ou égale à la précédente[79]. Pour être en mesure d'évaluer la condition de cette alternative simple, vous devrez utiliser une autre variable de type `TestInt` ; c'est à vous de décider comment en faire bon usage pour y parvenir. À chaque fois que cette condition est vérifiée il faut incrémenter un compteur dédié puis afficher le message `"Chuck Norris counted to infinity - "` suivi de `"once.\n"`, `"twice.\n"` ou `"NNNN times.\n"` (`NNNN` étant substitué par le nombre d'occurrences de ces messages). Pour distinguer ces trois cas vous utiliserez une alternative multiple (`switch-case`). Lorsque de tels messages auront été produits trois fois, il ne restera plus qu'à afficher `"then the numbers broke down!\n"` avant de provoquer la terminaison de la boucle.

Fabriquez à nouveau le programme et exécutez-le afin d'observer son comportement. Si ce programme se comporte mal, en restant bloqué dans la boucle par exemple, vous pouvez le terminer brutalement en saisissant la combinaison de touches `[Control]-[c]` dans le terminal depuis lequel vous l'avez lancé.

Reprenez l'expérience en utilisant le type `uint16_t` pour `TestInt` : les rebouclages doivent se produire bien plus rapidement. Si vous souhaitiez tenter l'expérience avec le type `uint64_t` il vous faudrait environ quatre milliards de fois plus longtemps qu'avec le type `uint32_t`[80].

Comme annoncé, cet exercice ne met pas en évidence de nouvelles propriétés ; il ne sert qu'à pratiquer en réutilisant des notions déjà présentées.

4.3.3. Dépassement dans un calcul arithmétique

Jusqu'alors, les dépassements de capacité ont été provoqués de manière évidente. Ce nouvel exemple montre qu'ils surviennent parfois de manière plus subtile.

Vous devez maintenant réaliser une fonction `intComputation()` qui attend trois paramètres entiers nommés `a`, `b` et `c` et ne renvoie aucun résultat. Elle sera déclarée dans le fichier d'en-tête `typeProperties.h` et définie dans le fichier `typeProperties.c`. Il faudra ajouter au programme principal du fichier `prog02.c` un appel à cette fonction avec les valeurs `2`, `3` et `4`. Cette nouvelle fonction doit elle aussi commencer par afficher son nom et la valeur de ses paramètres `a`, `b` et `c`.

Dans un premier temps, faites afficher clairement à cette fonction le résultat du calcul `a*b/c`.

Fabriquez à nouveau le programme et exécutez-le afin d'observer le résultat obtenu. Du point de vue arithmétique le résultat devrait être 1.5 mais puisque ce calcul n'utilise que des entiers aucune décimale n'est retenue : nous obtenons la valeur `1`.

Intervenez maintenant dans le programme principal pour modifier l'appel à la fonction `intComputation()`. Elle doit désormais utiliser les paramètres effectifs `2*i`, `3*i` et `4*i`, `i` étant un entier contrôlant une boucle `for`. Cet entier compte de `1` à `1000000` (un million) en se multipliant par `10` à chaque itération.

Fabriquez à nouveau le programme et exécutez-le afin d'observer la séquence de résultats obtenue. Si les premiers résultats semblent progresser de manière cohérente (des multiples de 1.5), ils deviennent aberrants (négatifs notamment) lorsque les paramètres `a`, `b` et `c` deviennent multiples de `100000` (cent mille) et au-delà. Pourtant, un simple calcul mental permet de prévoir que les résultats attendus tiennent aisément dans la plage de valeur couverte par le type `int` utilisé pour ces calculs.

L'apparition de tels résultats aberrants est probablement due à un débordement de capacité intervenant avant que la division par `c` n'ait lieu. Pour investiguer dans cette direction, complétons la fonction `intComputation()` en ajoutant un affichage clair du résultat du calcul `a*(b/c)` (qui doit être équivalent au précédent du point de vue arithmétique).

79 Puisque nous ne faisons qu'incrémenter ce ne devrait jamais être le cas, mais sait-on jamais...

80 Cette fois-ci, l'intervention de Chuck Norris serait pleinement justifiée.

Un relance de l'expérience doit indiquer que cette nouvelle formulation du calcul donne systématiquement le résultat `0`. En effet, si vous ajoutez de la même façon un affichage clair du calcul `b/c` et relancez à nouveau l'expérience vous constaterez sans surprise que la division entière d'un entier par un autre qui lui est supérieur donne systématiquement `0` ; la multiplication de `a` par ce résultat partiel ne peut être que nulle.

Nous retiendrons de cette investigation infructueuse que ***la division entière doit intervenir le plus tardivement possible dans le calcul*** pour éviter de faire disparaître une partie substantielle (voire l'intégralité) des informations contenues dans les entiers.

Pour confirmer le fait que les résultats aberrants proviennent de la multiplication, complétons à nouveau la fonction `intComputation()` pour lui ajouter un affichage clair du résultat du calcul `a*b`. Fabriquez à nouveau le programme et exécutez-le afin de vérifier que cette multiplication donne bien des résultats inattendus en phase avec ceux du calcul complet initial.

En effet, un simple calcul mental confirme le fait que les valeurs atteintes par le résultat de cette multiplication ne tiennent pas dans la plage de valeur couverte par le type `int` utilisé pour ces calculs.

Pour mener à bien ce calcul, nous n'avons pas d'autre choix que de recourir à des entiers de plus grande capacité. Complétez alors à nouveau la fonction `intComputation()` pour lui ajouter un affichage clair du résultat d'un calcul équivalent à `a*b/c` mais dans lequel nous forçons l'usage du type `int64_t` par l'utilisation d'un *cast* (à vous de le placer judicieusement). Le résultat de ce calcul sera donc de type `int64_t` et il faudra un autre *cast* pour le convertir à nouveau vers le type `int` attendu.

Une nouvelle fabrication et relance de ce programme doit enfin donner les résultats escomptés même lorsque les paramètres ont des valeurs élevées.

Malgré le fait que les paramètres d'un calcul respectent la plage de valeurs du type choisi, et qu'il en est de même pour le résultat théoriquement attendu, nous retiendrons de cet exemple qu'il est néanmoins possible que le résultat d'une opération intermédiaire provoque un débordement de capacité. Ceci est particulièrement fréquent concernant les multiplications. Il convient donc, dans le cadre d'un problème donné, de ***toujours s'interroger sur les plages de valeurs effectivement atteintes par les paramètres d'un calcul afin d'évaluer les risques de débordement de capacité lors des opérations intermédiaires***.

4.3.4. Promotion automatique des petits entiers

Lorsque nous travaillons avec de petits entiers (`char`, `short`, `int8_t`, `int16_t`...) le risque de débordement dans les calculs, tels que provoqués précédemment, sont très probables. Afin de les limiter, le langage C utilise une règle qui consiste à promouvoir en `int` (éventuellement `unsigned` selon le besoin) les entiers qui sont plus petits dès lors qu'on leur applique une opération (addition, multiplication...).

(http://en.cppreference.com/w/c/language/conversion , rubrique *Integer promotions*)

Afin de mettre ce comportement en évidence, vous devez maintenant réaliser une fonction `intPromotion()` qui n'attend aucun paramètre et ne renvoie aucun résultat. Elle sera déclarée dans le fichier d'en-tête `typeProperties.h` et définie dans le fichier `typeProperties.c`. Il faudra ajouter au programme principal du fichier `prog02.c` un appel à cette fonction. Cette nouvelle fonction doit elle aussi commencer par afficher son nom.

Dans cette fonction, déclarez trois variables `a8`, `b8` et `c8` comme des entiers de 8 *bits* ; initialisez les deux premiers avec des valeurs arbitraires et le troisième comme étant la somme des deux précédents. Affichez alors le résultat de l'opérateur `sizeof` appliqué à `c8` et à l'expression `a8+b8`.

Fabriquez à nouveau le programme et exécutez-le afin d'observer le résultat obtenu. À la compilation, si le compilateur est sévère, vous devez obtenir un message d'avertissement qui vous indique qu'une perte d'information est possible lors de l'initialisation de `c8` ; laissez-le pour l'instant et observez attentivement les résultats. Vous devez constater que, bien que `a8` et

b8 n'occupent qu'un octet chacun, l'opération qui les combine manipule des données aussi grandes qu'un *int* (voir les résultats de la fonction *displaySize()*) : la promotion a bien eu lieu.

Complétez cette démarche avec des variables *a16*, *b16* et *c16* sur 16 *bits*, *a32*, *b32* et *c32* sur 32 *bits*, *a64*, *b64* et *c64* sur 64 *bits*. Fabriquez à nouveau le programme et exécutez-le afin de constater que la promotion n'a bien lieu que pour les entier plus petits que *int*.

Finissez en ajoutant les conversions explicites (*cast*) qui permettent d'éliminer les messages d'avertissement sur l'initialisation de *c8* et *c16*.

4.4. Les limites des réels

Le langage C propose trois types pour représenter des valeurs réelles[81] : *float*, *double* et *long double*.

(http://en.cppreference.com/w/c/language/arithmetic_types)

Même si la forme de leur représentation n'est pas strictement spécifiée dans la norme du langage C, dans la pratique ils respectent très souvent la norme IEEE754.

(https://en.wikipedia.org/wiki/IEEE_floating_point)

Cette représentation repose sur un *bit* indiquant le signe (positif ou négatif), une mantisse (une série de *bits* valant chacun $1/2^N$) et un exposant (une puissance de deux positive ou négative) appliqué à cette mantisse. Observez quelques valeurs réelles avec cet outil :

(https://www.h-schmidt.net/FloatConverter/IEEE754.html)

4.4.1. L'arrondi de la représentation réelle

Puisque les types représentant une valeur réelle en langage C ne peuvent avoir une représentation de longueur infinie, cet exercice vise à mettre en évidence l'effet de l'arrondi qu'ils introduisent.

Vous devez maintenant réaliser une fonction *realRounding()* qui attend un paramètre de type *double* nommés *step* et ne renvoie aucun résultat. Elle sera déclarée dans le fichier d'en-tête *typeProperties.h* et définie dans le fichier *typeProperties.c*. Il faudra ajouter au programme principal du fichier *prog02.c* un appel à cette fonction avec la valeur *0.375*. Cette nouvelle fonction doit elle aussi commencer par afficher son nom et la valeur de son paramètre *step*.

Complétez le corps de cette fonction pour qu'elle contienne une boucle *for* faisant évoluer un réel de type *double* de *0.0* à *6.0* par incrément successifs de *step* (le paramètre) ; chaque itération devra afficher clairement la valeur de ce réel. Déclarez un autre réel nommé *three* et valant *3.0*. Si, lors d'une itération de la boucle, le réel servant à son contrôle est égal à *three*, alors affichez un message explicite et terminez prématurément la boucle par l'instruction *break*.

Fabriquez à nouveau le programme et exécutez-le afin d'observer la séquence des valeurs réelles affichée. Les incréments successifs de *0.375* finissent par atteindre exactement la valeur *3.0* ce qui met fin à la boucle : c'est bien le résultat attendu.

Intervenez maintenant dans le programme principal pour ajouter un nouvel appel à la fonction *realRounding()* en utilisant cette fois-ci la valeur *0.3* comme paramètre et relancez l'expérience. La nouvelle séquence de valeurs obtenue semble bien passer par la valeur *3.0* mais se poursuit sans s'interrompre jusqu'à approcher de *6.0*. Pourtant, d'un point de vue arithmétique, dix incréments de *0.3* donnent bien exactement la valeur *3.0* !

Dans le but d'investiguer autour de cette incohérence, complétons l'affichage produit dans la boucle pour lui faire également indiquer la différence entre la variable qui nous intéresse et la variable *three*. Une nouvelle relance de l'expérience doit faire apparaître un écart infinitésimal aux abords de *3.0*. L'affichage de la valeur *3.0* exacte n'est due qu'aux options de formatage

81 Le terme réel est abusif puisque la représentation n'est pas infinie ; il ne s'agit que de nombres à virgule flottante.

par défaut de la fonction *printf()* qui, pour un meilleur confort visuel, choisit de n'écrire qu'un nombre raisonnable de décimales ; la vraie valeur n'est donc pas strictement égale à *3.0* mais en diffère très légèrement.

Pour comprendre la raison de cet écart, raisonnons en base dix. La valeur 1/4 se représente exactement par 0.25 et quatre accumulations de cette valeur donnent bien le résultat 1.0. En revanche, une valeur comme 1/3 nécessiterait une infinité de décimales valant 3 pour être représentée exactement. Si nous n'en retenons arbitrairement que six, cette valeur est arrondie à 0.333333. Accumuler trois fois cette valeur donne le résultat 0.999999 qui est certes très proche de 1.0 mais n'y est toutefois pas strictement égal.

Un raisonnement similaire peut être tenu en base deux. Pour revenir à notre programme, la valeur 0.375 est une somme de puissances négatives de deux (1/4 et 1/8) et se représente donc dans la mantisse d'un réel par un motif binaire fini. Veuillez le vérifier avec cet outil :

(https://www.h-schmidt.net/FloatConverter/IEEE754.html)

Huit accumulations de ce motif fini donnent bien un autre motif fini valant exactement 3.0. En revanche, la valeur 0.3 se représente par un motif binaire infini ; il sera donc tronqué dans la mantisse. Veuillez le vérifier avec le même outil interactif. Dix accumulations de cette valeur produisent une propagation de la troncature et finissent par donner une valeur légèrement différente de 3.0.

Malgré le fait que les types réels du langage C permettent d'exprimer une très grande variété de valeurs à virgule flottante, nous retiendrons de cet exemple qu'il existe des valeurs qui ne peuvent pas être représentées exactement. L'accumulation des arrondis dans les calculs peut alors produire des résultats contre-intuitifs. Par conséquent, ***il est généralement incorrect de tester une égalité stricte sur des nombres réels puisque les valeurs attendues d'un point de vue arithmétique risquent de ne pas être obtenues en pratique ; il vaut mieux s'appuyer sur les inégalités***.

4.4.2. Les ordres de grandeur des valeurs réelles

Contrairement aux types entiers, les types réels du langage C offrent l'avantage de représenter des valeurs à la fois très grandes et très petites. Nous nous proposons ici d'explorer ces limites.

Vous devez maintenant réaliser une fonction *realLimits()* qui attend un paramètre de type *float* nommé *step* et ne renvoie aucun résultat. Elle sera déclarée dans le fichier d'en-tête *typeProperties.h* et définie dans le fichier *typeProperties.c*. Il faudra ajouter au programme principal du fichier *prog02.c* un appel à cette fonction avec la valeur *0.125f*[82]. Cette nouvelle fonction doit elle aussi commencer par afficher son nom et la valeur de son paramètres *step*. De plus, vous afficherez clairement les valeurs des constantes réelles *FLT_MIN*, *FLT_MAX*, *DBL_MIN* et *DBL_MAX* du fichier d'en-tête standard *float.h*.

(http://en.cppreference.com/w/c/types/limits)

Fabriquez à nouveau le programme et exécutez-le afin de constater les ordres de grandeur annoncés pour les types *float* et *double* : les valeurs annoncées sont assez impressionnantes, même dans le cas de réels en simple précision[83].

Complétez le corps de la fonction *realLimits()* pour qu'elle contienne une boucle *for* utilisant un compteur entier qui commence à *0* et qui est incrémenté à chaque itération ; cependant, cette boucle n'a pas de condition, elle se poursuit donc infiniment. À chaque itération, elle doit accumuler dans une variable *accum*, de type *float* et initialement nulle, la valeur du paramètre *step*. Vous tenterez de détecter (à l'aide d'une autre variable de type *float*) si, contre toute attente, la variable *accum* a la même valeur avant et après une accumulation. Dans ce cas, il faudra produire un message qui affiche clairement la valeur du compteur de boucle et de la variable *accum* avant de terminer la boucle par l'instruction *break*.

82 Le suffixe *f* accolé à une constante littérale réelle indique qu'il s'agit d'un *float* et non d'un *double*.

83 Les ordres de grandeur d'un atome d'hélium et du système solaire sont respectivement de 10^{-10} et 10^{13} mètres.

Fabriquez à nouveau le programme et exécutez-le afin d'en observer le comportement ; le cas inattendu est bien détecté.

Intervenez maintenant dans le programme principal pour ajouter deux nouveaux appels à la fonction `realLimits()` en utilisant cette fois-ci les valeurs `0.1f` et `0.5f` comme paramètre et relancez l'expérience. Vous devez constater que l'ordre de grandeur du nombre d'itérations au bout duquel apparaît le phénomène est le même dans les trois cas : il est même égal pour les incréments `0.125` et `0.5` qui sont tous les deux des puissances négatives de deux. Les valeurs obtenues par accumulation sont quant à elles dépendantes de l'incrément choisi : `0.125` et `0.1` donnent des résultats comparables alors que `0.5` donne un résultat quatre fois plus grand que `0.125`.

L'anomalie d'accumulation détectée ici est d'autant plus surprenante que la valeur de l'incrément et celle de la valeur accumulée sont bien loin des limites connues pour le type `float`. L'explication tient à nouveau dans la troncature de la mantisse des réels. Si en base dix nous souhaitons ajouter l'incrément 0.1 à un réel valant déjà un million, tout en ne conservant que six décimales, le résultat sera représenté par $1.000000*10^6$: l'incrément est perdu car il figure au-delà des décimales retenues.

Un raisonnement similaire peut être tenu avec la représentation binaire de la mantisse de notre variable `accum`. Elle n'est constituée que de vingt-quatre *bits*[84], donc l'ajout d'un incrément à une valeur qui est plus de 2^{24} fois plus grande ne peut en aucun cas modifier cette dernière. Pour vous en convaincre, confrontez donc le nombre d'itérations constatées pour les incréments `0.125` et `0.5` à cette grandeur. Remarquez que le même phénomène peut se produire sur une variable de type `double` ; toutefois, l'échéance sera bien plus tardive puisque la mantisse est constituée de cinquante-trois *bits*.

Malgré le fait que les types réels du langage C peuvent recouvrir des valeurs allant de l'infiniment grand à l'infiniment petit, nous retiendrons de cet exemple que ***le calcul sur des réels dont les valeurs relatives sont extrêmement éloignées donne très probablement un résultat aberrant***. Ceci est particulièrement flagrant avec le type `float`, c'est pourquoi nous lui préférons en général le type `double`[85].

Par extension, le calcul de la différence entre deux valeurs réelles très proches l'une de l'autre ne sera exprimé qu'avec une très mauvaise précision (seuls quelques *bits* à la fin de leurs mantisses respectives diffèrent).

4.4.3. Application au développement en série

Nous réutilisons ici, à une fin d'entraînement, les propriétés constatées sur les réels vis-à-vis des très faibles incréments pour approximer la fonction mathématique de logarithme naturel par le développement en série suivant :

$$\forall x \in]0,+\infty[\quad \ln(x) = \sum_{k=0}^{\infty} \frac{2}{2k+1} \left(\frac{x-1}{x+1}\right)^{2k+1}$$
$$= 2\left(\frac{x-1}{x+1}\right) + \frac{2}{3}\left(\frac{x-1}{x+1}\right)^3 + \frac{2}{5}\left(\frac{x-1}{x+1}\right)^5 + \frac{2}{7}\left(\frac{x-1}{x+1}\right)^7 + \dots$$

Il s'agit en effet d'accumuler des incréments de plus en plus faibles au fur et à mesure que l'ordre du développement augmente. Le choix préalable et arbitraire d'un ordre maximal introduirait un arrondi du résultat alors que le réel serait en mesure de représenter plus précisément le résultat théorique attendu. Nous procéderons donc comme dans l'exercice précédent en itérant jusqu'à détecter un incrément ne produisant plus d'effet sur la valeur accumulée : la valeur recherchée aura ainsi été atteinte avec la précision maximale.

Vous devez pour cela réaliser une fonction `computeLog()` qui attend en paramètre un réel de type `double` nommé `x` et qui renvoie un résultat du même type. Elle sera déclarée dans le fichier d'en-tête `typeProperties.h` et définie dans le fichier `typeProperties.c`. Il faudra

84 Un *bit* est implicitement à un et les vingt-trois suivants sont explicitement représentés.

85 Il existe quelques architectures matérielles sur lesquelles les calculs en double-précision sont plus lents que ceux en simple-précision mais sur beaucoup d'autres la seule différence ne concerne que la taille occupée en mémoire.

ajouter au programme principal du fichier `prog02.c` une boucle d'appels à cette fonction pour des valeurs réelles allant de `0.2` à `3.0` par pas de `0.2`. À chaque itération il faudra afficher clairement le paramètre et le résultat afin de contrôler *a posteriori* les valeurs obtenues.

La fonction `computeLog()` sera dans sa structure très semblable à la fonction `realLimits()` : seul le calcul de l'incrément nécessitera une attention particulière. Il est recommandé de calculer une fois pour toutes, avant d'itérer, le terme `s=(x-1)/(x+1)` ; en effet, sa valeur restera inchangée tout au long du calcul. Il sera facile de partir d'un premier terme valant `t=2.0*s` puis d'itérer à partir de `k=1` ; ainsi, à chaque nouvelle itération il suffit de multiplier le terme `t` par le carré de `s`. L'incrément se déduit de chaque terme `t` en le divisant par `2*k+1`. Lorsque l'incrément n'a plus d'effet sur la valeur accumulée, il suffit de quitter la boucle avec l'instruction `break`.

Fabriquez à nouveau le programme et exécutez-le afin d'observer les valeurs produites par ce calcul. Vérifiez que les valeurs obtenues sont correctes en vous aidant d'une calculette ou simplement de l'interpréteur Python (fonction `math.log()`).

4.4.4. Digression : étude de l'évolution des variables au débogueur

Nous profitons de l'exercice d'entraînement précédent pour nous familiariser avec quelques fonctionnalités du débogueur : il s'agit d'observer l'évolution des variables.

Le fichier `GNUmakefile` qui vous a été fourni spécifie déjà l'usage de l'option `-g` lors de l'invocation du compilateur. Au lancement de l'outil de débogage :

```
$ tdb prog02 ↲
```

notre programme est initialement en pause ; la toute première ligne de la fonction main() est mise en surbrillance pour indiquer qu'il s'agit de la prochaine ligne de code qui sera exécutée.

Nous procéderons à l'investigation selon les étapes suivantes :

- un *double-clic* dans la zone d'affichage du code source, juste sur la ligne qui provoque l'appel à la fonction `computeLog()`, place un point-d'arrêt matérialisé par un changement de couleur,
- le bouton `Continue [F8]` (triangle) permet de lancer l'exécution du programme,
  - tout le début du programme s'exécute normalement puis la zone de messages indique que le programme est arrêté sur le point-d'arrêt,
- un nouveau *double-clic* sur la même ligne annule ce point-d'arrêt,
- le bouton `Step [F5]` (entrée dans les accolades) permet d'atteindre la prochaine ligne de code dans le programme,
  - nous sommes désormais sur la première ligne de la fonction `computeLog()` et la zone de messages nous indique la valeur de son paramètre,
- un *double-clic* sur la ligne qui suit le calcul de l'incrément à l'intérieur de la boucle place un nouveau point-d'arrêt,
- le bouton `Continue [F8]` relance l'exécution,
  - la zone de messages nous indique que nous avons atteint ce nouveau point d'arrêt,
- après avoir surligné le compteur de boucle `k`, le bouton `Show selected variable` (paire de lunettes) affiche dans la zone de message la valeur de cette variable,
  - il est tout aussi simple de saisir `print k` ou encore `p k` sur la ligne de commande située au bas de l'outil de débogage,
- procédez de la même façon pour afficher la valeur de `x` et de l'incrément,
  - le menu `Show/Locals` produit l'affichage de toutes les variables locales,
- le bouton `Continue [F8]` relance l'exécution,
  - la zone de messages nous indique que nous avons atteint le même point d'arrêt,
- reprenez l'affichage des variables pour constater que nous avons effectué une nouvelle itération,
- l'usage de la commande `display` (ou du bouton) en lieu et place de la commande `print` provoquera l'affichage des variables concernées à chaque arrêt,

    - utilisez le bouton *Continue [F8]* plusieurs fois pour constater l'enchaînement des affichages automatiques dans la zone de messages,
        - les itérations sont nombreuses lorsque *x* vaut *0.2*,
- un nouveau *double-clic* sur la ligne du point-d'arrêt annule celui-ci,
- saisissez *break NUM if x>1.1* sur la ligne de commande située au bas de l'outil de débogage en remplaçant *NUM* par le numéro de ligne de l'ancien point-d'arrêt,
    - ce point-d'arrêt ne sera bloquant que si la condition est respectée,
- le bouton *Continue [F8]* relance l'exécution,
    - plusieurs invocations de la fonction *computeLog()* ont lieu,
    - la zone de messages nous indique que nous avons atteint le point d'arrêt et affiche les variables sélectionnées avec la commande *display*,
- utilisez le bouton *Continue [F8]* plusieurs fois pour constater l'enchaînement des affichages automatiques dans la zone de messages,
    - les itérations sont peu nombreuses lorsque *x* vaut *1.2*,
- nous quittons la session de débogage en fermant simplement la fenêtre de l'outil.

L'utilisation qui a été faite du débogueur ici visait à illustrer la démarche très classique qui consiste à inspecter l'état des variables au fil de l'exécution du programme. Nous avons notamment fait usage d'un point-d'arrêt conditionnel afin de ne bloquer l'exécution que lorsque les variables sont dans un état propice à une observation que nous souhaitons mener. Ceci peut aider à la mise au point d'un programme en recherchant la cause qui est à l'origine de la production d'un résultat inattendu.

4.5. Résumé

Ici s'achève cette deuxième séance pratique dans laquelle nous nous sommes familiarisés avec la variété de types entiers et réels disponibles dans le langage C. Nous avons mis en évidence quelques propriétés usuelles qui permettent d'effectuer des choix lorsque les spécificités du problème abordé l'imposent.

Voici un récapitulatif des points à retenir et à maîtriser :

- l'opérateur *sizeof* permet de connaître la taille en octets d'un type,
- les types entiers ont des tailles qui varient d'une plateforme d'exécution à une autre,
- il existe toutefois des entiers dont on choisit précisément la taille de manière portable,
- les entiers représentent une plage de valeurs limitée qui dépend de leur taille (des *macros* permettent d'en prendre connaissance),
- le type entier à utiliser en priorité est le type *int* (les autres ne répondent qu'à des besoins très spécifiques concernant les plages de valeurs),
- le débordement de capacité sur un entier non-signé (*unsigned*) donne un résultat défini (le rebouclage vers l'autre extrémité de la plage de valeurs),
- le débordement de capacité sur un entier signé donne un résultat indéfini (il faut donc le préférer à un non-signé si aucun rebouclage n'est envisagé),
- un débordement de capacité peut se produire sur les étapes intermédiaires d'un calcul sur des entiers, notamment lors d'une multiplication,
- une division entière ne doit intervenir que le plus tardivement possible dans un calcul,
- les valeurs réelles comportent des arrondis,
- le test d'égalité stricte entre des valeurs réelles est généralement incorrect et doit être évité,
- le calcul sur des réels dont les valeurs relatives son extrêmement éloignées donne très probablement un résultat aberrant,
- le type réel à utiliser en priorité est le type *double* (le type *float* est bien plus sujet aux problèmes de précision relative).

Pour vous entraîner, et en vue de passer l'épreuve pratique, veuillez réaliser un programme qui manipule différents types d'entiers et de réels en utilisant des pointeurs et des tableaux (découverts dans le cours du chapitre 3) pour calculer quelques traitements simples (somme, moyenne…).

Voici le contenu des fichiers réalisés dans ce sujet, tels qu'ils doivent être si vous avez suivi scrupuleusement les recommandations qui vous ont été données à chaque étape.

```
//---- prog02.c ----
#include "typeProperties.h"

int
main(void)
{
displaySize();
intOverflow(10000);
intOverflow(100000);
tributeToChuck();
for(int i=1;i<=1000000;i*=10)
  { intComputation(2*i,3*i,4*i); }
realRounding(0.375);
realRounding(0.3);
realLimits(0.125f);
realLimits(0.1f);
realLimits(0.5f);
printf("\n");
for(double x=0.2;x<3.0;x+=0.2)
  { printf("computeLog(%g)=%g\n",x,computeLog(x)); }
return 0;
}
```

```
//---- typeProperties.h ----
#ifndef TYPEPROPERTIES_H
#define TYPEPROPERTIES_H 1

void
displaySize(void);

void
intOverflow(int limit);

void
tributeToChuck(void);

void
intComputation(int a, int b, int c);

void
intPromotion(void);

void
realRounding(double step);

void
realLimits(float step);

double
computeLog(double x);

#endif // TYPEPROPERTIES_H
```

```
//---- typeProperties.c ----
#include "typeProperties.h"
#include <stdio.h>
#include <stdbool.h>
#include <stdint.h>
#include <limits.h>
#include <float.h>

void
displaySize(void)
{
printf("\n~~~~ displaySize() ~~~~\n");
printf("sizeof(size_t)      = %d\n",(int)sizeof(size_t)     );
printf("sizeof(bool)        = %d\n",(int)sizeof(bool)       );
printf("sizeof(char)        = %d\n",(int)sizeof(char)       );
printf("sizeof(short)       = %d\n",(int)sizeof(short)      );
printf("sizeof(int)         = %d\n",(int)sizeof(int)        );
printf("sizeof(long)        = %d\n",(int)sizeof(long)       );
printf("sizeof(long long)   = %d\n",(int)sizeof(long long)  );
printf("sizeof(float)       = %d\n",(int)sizeof(float)      );
printf("sizeof(double)      = %d\n",(int)sizeof(double)     );
printf("sizeof(long double) = %d\n",(int)sizeof(long double));
printf("sizeof(int8_t)      = %d\n",(int)sizeof(int8_t)     );
printf("sizeof(int16_t)     = %d\n",(int)sizeof(int16_t)    );
printf("sizeof(int32_t)     = %d\n",(int)sizeof(int32_t)    );
printf("sizeof(int64_t)     = %d\n",(int)sizeof(int64_t)    );
printf("sizeof(uint8_t)     = %d\n",(int)sizeof(uint8_t)    );
printf("sizeof(uint16_t)    = %d\n",(int)sizeof(uint16_t)   );
printf("sizeof(uint32_t)    = %d\n",(int)sizeof(uint32_t)   );
printf("sizeof(uint64_t)    = %d\n",(int)sizeof(uint64_t)   );
}

void
intOverflow(int limit)
{
printf("\n~~~~ intOverflow(%d) ~~~~\n",limit);
for(int i=0;i<limit;++i)
  {
  // uint16_t other=(uint16_t)i;
  int16_t other=(int16_t)i;
  if(other!=i)
    {
    printf("%d was turned into %d\n",i,other);
    break;
    }
  }
printf("UINT16_MAX = %d\n",UINT16_MAX);
printf("INT16_MIN  = %d\n",INT16_MIN);
printf("INT16_MAX  = %d\n",INT16_MAX);
printf("INT_MIN    = %d\n",INT_MIN);
printf("INT_MAX    = %d\n",INT_MAX);
printf("CHAR_MIN   = %d\n",CHAR_MIN);
printf("CHAR_MAX   = %d\n",CHAR_MAX);
}



                                                                    // ...(1/3)...
```

```
                                                                    // ...(2/3)...
void
tributeToChuck(void)
{
printf("\n~~~~ tributeToChuck() ~~~~\n");
typedef uint16_t TestInt;
int loopCount=0;
TestInt prev=0;
for(TestInt i=0;;++i)
  {
  if(i<=prev)
    {
    ++loopCount;
    printf("Chuck Norris counted to infinity - ");
    switch(loopCount)
      {
      case 1:  { printf("once.\n");               break; }
      case 2:  { printf("twice.\n");              break; }
      default: { printf("%d times.\n",loopCount); break; }
      }
    if(loopCount>2)
      { printf("then the numbers broke down!\n"); break; }
    }
  prev=i;
  }
}

void
intComputation(int a, int b, int c)
{
printf("\n~~~~ intComputation(%d,%d,%d) ~~~~\n",a,b,c);
printf("a*b/c = %d\n",a*b/c);
printf("a*(b/c) = %d\n",a*(b/c));
printf("b/c = %d\n",b/c);
printf("a*b = %d\n",a*b);
printf("(int)((int64_t)a*b/c) = %d\n",(int)((int64_t)a*b/c));
}

void
intPromotion(void)
{
printf("\n~~~~ intPromotion() ~~~~\n");
int8_t a8=3, b8=7, c8=(int8_t)(a8+b8);
printf("sizeof(c8)=%d sizeof(a8+b8)=%d\n",
       (int)sizeof(c8),(int)sizeof(a8+b8));
int16_t a16=3, b16=7, c16=(int16_t)(a16+b16);
printf("sizeof(c16)=%d sizeof(a16+b16)=%d\n",
       (int)sizeof(c16),(int)sizeof(a16+b16));
int32_t a32=3, b32=7, c32=a32+b32;
printf("sizeof(c32)=%d sizeof(a32+b32)=%d\n",
       (int)sizeof(c32),(int)sizeof(a32+b32));
int64_t a64=3, b64=7, c64=a64+b64;
printf("sizeof(c64)=%d sizeof(a64+b64)=%d\n",
       (int)sizeof(c64),(int)sizeof(a64+b64));
}

                                                                    // ...(2/3)...
```

```
                                                                    // ...(3/3)...
void
realRounding(double step)
{
printf("\n~~~~ realRounding(%g) ~~~~\n",step);
double three=3.0;
for(double d=0.0;d<6.0;d+=step)
  {
  printf("d=%g\tdelta=%g\n",d,d-three);
  if(d==three)
    { printf("found three!\n"); break; }
  }
}

void
realLimits(float step)
{
printf("\n~~~~ realAccumulation(%g) ~~~~\n",step);
printf("FLT_MIN = %g\n",FLT_MIN);
printf("FLT_MAX = %g\n",FLT_MAX);
printf("DBL_MIN = %g\n",DBL_MIN);
printf("DBL_MAX = %g\n",DBL_MAX);
float accum=0.0f;
for(int i=0;;++i)
  {
  float prev=accum;
  accum+=step;
  if(prev==accum)
    {
    printf("stalled after %d steps (%g)\n",i,accum);
    break;
    }
  }
}

double
computeLog(double x)
{
double s=(x-1.0)/(x+1.0);
double t=2.0*s;
double y=t;
double s2=s*s;
for(int k=1;;++k)
  {
  t*=s2;
  double delta=t/(2*k+1);
  double prev=y;
  y+=delta;
  if(prev==y)
    {
    // printf("x=%g  k=%d  y=%g  delta=%g\n",x,k,y,delta);
    break;
    }
  }
return y;
}
```

5. C03_Strings : Chaînes de caractères

Jusqu'alors, l'essentiel des types de données présentés est dédié à un usage arithmétique ou logique (booléens, entiers, réels et leurs variantes). Les pointeurs et les tableaux font exception à cette tendance mais ne représentent finalement qu'un moyen indirect de manipuler les types précédents. Dans le cours d'introduction (chapitre 1), il est toutefois mentionné le type `char` qui est usuellement utilisé pour représenter des caractères. Nous verrons qu'il n'existe pas à strictement parler de type pour représenter les chaînes de caractères en langage C, mais des facilités pour considérer des séquences de `char` en tant que telles.

5.1. Séquences de caractères

L'usage commun d'une ***chaîne de caractères*** (vocabulaire) consiste à représenter et manipuler un texte quelconque. Celui-ci peut être de longueur nulle, ne contenir qu'un seul caractère, représenter un mot, ou bien une phrase avec de la ponctuation et pourquoi pas un paragraphe, une page ou encore un document complet. Cela marque une différence fondamentale avec les types du langage C que nous connaissons : une chaîne de caractère n'occupe probablement pas le même nombre d'octets en mémoire selon la valeur qu'elle a. Il ne s'agit donc pas d'un type de base, à la taille fixe sur une plateforme d'exécution, mais au contraire d'un type élaboré pour représenter une séquence d'information de longueur variable.

5.1.1. Le type des caractères

Le type `char` (déjà présenté en 1.3.2) n'est rien d'autre qu'un petit entier tenant sur un seul octet. Selon la plateforme d'exécution, il peut être implicitement signé ou non-signé. Si nous souhaitions l'utiliser pour des calculs arithmétiques (ce qui est possible), il nous faudrait probablement le qualifier de `signed` ou `unsigned` afin de contrôler précisément ses limites (voir l'exercice pratique du chapitre 4). Un usage plus courant de ce type consiste à interpréter sa valeur numérique comme un caractère (d'où son nom). La conversion entre cette valeur numérique et un caractère n'est qu'affaire de convention. Il existe à ce propos une table de conversion unanimement reconnue : la table ***ASCII*** (vocabulaire).

(http://en.cppreference.com/w/c/language/ascii)

Un périphérique spécialisé (imprimante, écran...) qui reçoit une telle valeur numérique tenant sur un octet doit réagir en produisant l'effet approprié. Si le code reçu représente un caractère imprimable, alors un dessin qui évoque aux êtres humains une lettre, un chiffre ou de la ponctuation (éventuellement vide si c'est une espace) sera produit. Il existe également des codes représentant des caractères de contrôle, c'est à dire qui provoquent la modification de la position d'écriture (changer de ligne, avancer jusqu'à prochain point de tabulation, etc).

Voici, à titre d'exemple, l'effet produit par la réception de tels caractères dans le terminal :

```
char txt[10]={ 72, 105, 44, 9, 69, 78, 73, 66, 33, 10 }; // saisie des codes ASCII
for(int i=0;i<10;++i) { printf("%c",txt[i]); }  // Hi,     ENIB!
for(int i=0;i<10;++i) { printf("%d ",txt[i]); } // 72 105 44 9 69 78 73 66 33 10
```

Remarquez que le format `"%c"` utilisé dans `printf()` transmet tel quel le code numérique du caractère vers le terminal (vérifiez avec la table ASCII). Le format `"%d"` traduit au contraire la

valeur entière de ce code numérique en une séquence de codes dont la réception par le terminal provoque l'affichage de caractères ayant l'apparence de chiffres.

(http://en.cppreference.com/w/c/io/fprintf)

5.1.2. Les caractères littéraux

Fort heureusement, les programmeurs n'ont pas à connaître tous ces codes numériques pour produire des message intelligibles. Le compilateur du langage C a connaissance de la table ASCII et produit le code numérique attendu. Il nous suffit de saisir, dans le code source du programme, le caractère désiré entre une paire de symboles `'` (apostrophe).

(http://en.cppreference.com/w/c/language/character_constant)

Voici une formulation alternative, mais strictement équivalente, de l'exemple précédent :

```
char txt[10]={ 'H', 'i', ',', '\t', 'E', 'N', 'I', 'B', '!', '\n' }; // saisie littérale
for(int i=0;i<10;++i) { printf("%c",txt[i]); }  // Hi,     ENIB!
for(int i=0;i<10;++i) { printf("%d ",txt[i]); } // 72 105 44 9 69 78 73 66 33 10
```

Il est important de comprendre que la notation utilisant des apostrophes n'est qu'une façon alternative de saisir une valeur entière littérale dans le code source. Ainsi `'A'`, `65` ou encore `0x41` représentent tous exactement la même valeur, et `'A'+10` vaut `75` tout comme `'K'`. Les caractères non imprimables peuvent être saisis à l'aide du symbole `\` (backslash) : par exemple `'\n'` pour changer de ligne ou `'\t'` pour insérer une tabulation.

(http://en.cppreference.com/w/c/language/escape)

Vous remarquerez certainement que la table ASCII ne décrit que les codes de `0` à `127` alors que l'octet constituant un `char` peut désigner deux fois plus de valeurs distinctes. Les codes supplémentaires ne sont pas standards et peuvent parfois désigner des caractères accentués qui varient d'une culture à une autre[86].

5.1.3. Les catégories de caractères

Il est mentionné plus haut que dans la table ASCII existent des caractères imprimables et des caractères de contrôle. Des fonctions standards permettent de les distinguer.

(http://en.cppreference.com/w/c/string/byte/isprint)

(http://en.cppreference.com/w/c/string/byte/iscntrl)

Comme indiqué sur ces pages de documentation, le fichier d'en-tête standard `ctype.h` propose encore bien d'autres fonctions pour distinguer plus finement ces catégories.

```
char txt[7]={ '1', ' ', 'M', 'o', 'R', 'e', '?' };
for(int i=0;i<7;++i) { if(isdigit(txt[i])) { printf("<%c>",txt[i]); } }          // <1>
for(int i=0;i<7;++i) { if(ispunct(txt[i])) { printf("<%c>",txt[i]); } }          // <?>
for(int i=0;i<7;++i) { if(isupper(txt[i])) { printf("<%c>",tolower(txt[i])); } } // <m><r>
```

Remarquez également que les fonctions `toupper()` et `tolower()` permettent de basculer les caractères alphabétiques vers les majuscules ou les minuscules.

En observant la table ASCII et la documentation de ces fonctions, vous constaterez que certaines catégories de caractères occupent des codes numériques contigus (pas toutes). Cela donne parfois l'opportunité, sur les problèmes qui le méritent, de déduire certains caractères par des relations arithmétiques.

86 L'étude des différents façons de représenter les jeux de codes de caractères dépasse de très loin l'objectif de cette prise en main du langage C.

```
char txt[12]={ 'B', 'h', 'e', ' ', 'F', 'r', 'p', 'e', 'r', 'g', '!', '\n' };
for(int i=0;i<12;++i)
  {
  char c=txt[i];
  if(isalpha(c))
    {
    int first=isupper(c) ? 'A' : 'a';
    c=(char)(first+((c-first)+13)%26);
    }
  printf("%c",c); // Our Secret!
  }
```

Cet exemple exploite cette propriété en transformant l'écart entre un caractère alphabétique et la lettre `'A'` ou `'a'` selon l'algorithme *ROT13*[87] (simulez-le à la main en vous aidant de la table ASCII). Nous renforçons ainsi la mise en évidence du fait que les caractères littéraux ne sont rien d'autre que des valeurs entières tenant sur un octet.

5.1.4. Le zéro terminal

Pour une raison de facilité, les exemples précédents stockaient les caractères dans un tableau afin de les énumérer dans une boucle. L'énumération de ces caractères pour produire un affichage revient finalement à afficher une chaîne de caractères. Nous pouvons donc assimiler un tableau de caractères à une chaîne. L'inconvénient de cette solution vient du fait qu'il faut connaître à l'avance le nombre d'éléments à énumérer dans ce tableau : ce nombre varie selon la longueur du texte à représenter.

Une convention unanimement adoptée consiste à utiliser un caractère de contrôle spécial pour indiquer que la chaîne est terminée. Ainsi, l'énumération des caractères d'une chaîne n'est pas simplement conditionnée par un compteur que l'on confronte à une limite préalablement déterminée, mais par les caractères découverts lors de cette énumération. Le caractère de contrôle en question est dénommé le ***caractère nul*** (ou zéro terminal, vocabulaire), il a le code ASCII `0` et est usuellement représenté de manière littérale par `'\0'`.

```
char txt[]={ 'H', 'e', 'l', 'l', 'o', ',', ' ', 'E', 'N', 'I', 'B', '!', '\n', '\0' };
for(int i=0;txt[i];++i) { printf("%c",txt[i]); } // Hello, ENIB!
```

Selon cette convention, la boucle d'énumération des caractères s'arrête dès qu'un caractère nul est rencontré ; il n'est donc pas nécessaire ici de déterminer explicitement le nombre d'éléments du tableau. Cette chaîne occupe un octet de plus que le nombre de ses caractères utiles puisqu'on y ajoute le caractère nul. Cette convention impose qu'une telle chaîne ne peut pas contenir `'\0'` parmi ses caractères utiles car tous les éventuels caractères qui le suivraient seraient forcément ignorés. Remarquez que lors de l'affichage d'une chaîne, le caractère nul n'est pas envoyé au terminal : la boucle est immédiatement terminée lors de sa détection[88].

L'usage du caractère de fin est tellement commun que bon nombre de fonctions qui traitent des chaînes en présupposent la présence et reposent sur sa détection. C'est le cas par exemple de la fonction `printf()` qui utilise le format `"%s"` pour énumérer automatiquement les caractères d'une chaîne jusqu'au zéro terminal et les afficher.

```
char txt[]={ 'H', 'e', 'l', 'l', 'o', ',', ' ', 'E', 'N', 'I', 'B', '!', '\0' };
printf("<%s>\n",txt); } // <Hello, ENIB!>
```

Le type attendu est `const char *`, ce qui est cohérent avec le fait que l'évaluation du nom d'un tableau fournit un pointeur sur son premier élément.

(http://en.cppreference.com/w/c/io/fprintf)

87 C'est un cas particulier du *chiffre de César* : un algorithme de chiffrement simpliste.

88 Ceci ne pose aucun problème dans la pratique dans la mesure où il ne doit produire aucun effet dans le terminal.

5.2. Initialisation des chaînes de caractères

La spécification de chaque élément du tableau sous la forme de caractères littéraux est certes plus aisée que l'usage des codes ASCII mais est encore loin d'être naturelle. Bien qu'il n'existe pas vraiment de type spécifiquement dédié à leur représentation, les chaînes terminées par un caractère nul sont d'un usage tellement courant que le langage C offre des facilités pour les initialiser avec des constantes littérales.

5.2.1. Les chaînes littérales

La notation utilisée pour spécifier la chaîne de format de la fonction *printf()* n'est rien d'autre qu'une chaîne littérale. Nous la reconnaissons au fait qu'elle encadre un texte par une paire de symboles " (guillemets). Il s'agit essentiellement d'une facilité d'écriture pour énumérer un à un les caractères littéraux de ce texte en y ajoutant le zéro terminal comme nous le faisions dans les exemples en 5.1.4.

```
char txt1[]={ 'H', 'e', 'l', 'l', 'o', ',', ' ', 'E', 'N', 'I', 'B', '!', '\0' };
char txt2[]="Hello, ENIB!";
printf("<%s><%s>\n",txt1,txt2); // <Hello, ENIB!><Hello, ENIB!>
```

Les deux formes d'initialisation des chaînes *txt1* et *txt2* sont strictement équivalentes : les guillemets se substituent à la liste d'initialisation du tableau. Les octets qui constituent ces deux chaînes sont bien stockés dans la pile comme tout tableau de classe de stockage automatique. Notez que cette facilité d'écriture n'existe que pour les tableaux de *char* ; pour les autres types, la liste d'initialisation est le seul moyen de spécifier le contenu initial du tableau. De la même façon que la liste d'initialisation ne peut pas servir à affecter un tableau après sa déclaration, ***une chaîne littérale ne peut être affectée à un tableau de*** *char* ***qu'au moment de sa déclaration***.

Puisque les chaînes littérales facilitent la saisie, il peut être tentant de saisir de longs textes. Seulement, pour préserver la lisibilité du code source nous n'avons pas intérêt à en allonger les lignes exagérément. Pour encore plus de facilité de saisie, le compilateur considère plusieurs chaînes littérales qui se suivent directement comme n'en formant qu'une seule.

```
char txt1[]="Only" " one " "string";
char txt2[]="This is a very long text that would hardly fit the width of"
            " my text editor! Fortunately, the designers of C thought about "
            "every detail.";
printf("<%s><%s>\n",txt1,txt2); // <Only one string><This is a very long text ... every detail.>
```

Le zéro terminal n'est placé qu'à la fin de la dernière des chaînes littérales successives.

5.2.2. Affectation à un pointeur

Le cours sur les pointeurs et les tableaux (chapitre 3) indiquait qu'il existe une conversion implicite donnant un pointeur à partir du nom d'un tableau, et que ce pointeur désigne l'adresse du premier élément de ce tableau. Il est donc correct d'initialiser un pointeur sur un *char* à partir d'un tableau de *char* représentant une chaîne.

```
char txt[]="Hello, ENIB!";
char *split=txt+5;
for(char *p=txt;p<split;++p)
  { *p=(char)toupper(*p); }     // passer les cinq premières lettres en majuscule
*(split++)='\0';                // marquer la fin juste après puis avancer
printf("<%s><%s>\n",txt,split); // <HELLO>< ENIB!>
```

À titre de révision, cet exemple utilise l'arithmétique des pointeurs pour modifier le contenu du tableau. Remarquez l'usage de l'opérateur de post-incrémentation qui incrémente le pointeur seulement après en avoir extrait l'adresse qui servira au déréférencement. En plaçant un caractère nul à la place de la virgule nous avons séparé la chaîne initiale en deux. La fonction *printf()* utilise tout aussi bien le nom d'un tableau qu'un pointeur pour afficher des chaînes.

Il est également possible d'affecter un pointeur depuis une chaîne littérale, mais cette opération n'a plus du tout la même signification que précédemment ! ***Les chaînes littérales qui ne servent pas à initialiser un tableau de*** `char` ***sont stockées dans une zone mémoire constante***. Leur valeur est l'adresse de leur premier caractère et doit être uniquement affectée à un pointeur vers des caractères constants[89]. En effet, une tentative d'écriture à cette adresse provoquerait un plantage du programme sur la plupart des plateformes d'exécution.

```
const char *txt="Hello, ENIB!";
printf("<%s><%s>\n",txt,txt+7); // <Hello, ENIB!><ENIB!>
```

Dans ces conditions, nous ne pouvons que consulter ces chaînes littérales. Remarquez que la chaîne de format que nous utilisons pour les invocations de `printf()` dans nos exemples est justement une chaîne littérale placée dans une zone mémoire constante ; elle est affectée au premier paramètre formel de cette fonction qui est bien un `const char *`.

(http://en.cppreference.com/w/c/io/fprintf)

Puisque les chaînes littérales ont été initialement présentées comme une facilité nous évitant d'avoir à saisir un à un les caractères littéraux d'un tableau, il est légitime de s'interroger sur la validité de cette équivalence lorsqu'on à affaire à la mémoire constante.

```
char txt1[]="correct";
char txt2[]={ 'c', 'o', 'r', 'r', 'e', 'c', 't', '\0' };
const char *txt3="correct";
// const char *txt4={ 'i', 'n', 'c', 'o', 'r', 'r', 'e', 'c', 't', '\0' }; // INCORRECT
```

L'échec à l'initialisation de `txt4` montre que l'équivalence n'est pas valide dans ce cas : les chaînes littérales sont vraiment traitées comme un cas à part !

5.3. Les fonctions de manipulation de chaînes

L'usage de chaînes de caractères à zéro terminal est tellement commun que le fichier d'en-tête standard `string.h` est justement dédié à la déclaration de fonctions spécialisées dans leur manipulation. Il s'agit d'un ensemble très vaste ; nous ne présentons ici que les plus courantes.

(http://en.cppreference.com/w/c/string/byte)

5.3.1. Longueur d'une chaîne

La longueur d'une chaîne se définit simplement par le nombre de caractères qui précèdent le zéro terminal[90]. Le nombre minimal d'octets nécessaire à la mémorisation d'une chaîne correspond donc à sa longueur plus un : un octet par caractère plus le zéro terminal.

La fonction standard `strlen()` est dédiée à ce propos.

(http://en.cppreference.com/w/c/string/byte/strlen)

Elle compte les caractères à partir de celui qui lui est transmis par pointeur jusqu'à rencontrer le caractère nul, qui n'est pas compté.

```
char str1[]="first string";
char str2[100]="second string";
const char *str3="third string";
printf("<%s> %d %d\n",str1,  (int)strlen(str1),  (int)sizeof(str1));   // <first string> 12 13
printf("<%s> %d %d\n",str2,  (int)strlen(str2),  (int)sizeof(str2));   // <second string> 13 100
printf("<%s> %d %d\n",str3,  (int)strlen(str3),  (int)sizeof(str3));   // <third string> 12 8
printf("<%s> %d %d\n",str2+7,(int)strlen(str2+7),(int)sizeof(str2+7)); // <string> 6 8
```

89 Avant que le mot-clef `const` ne soit introduit dans le langage, un simple `char *` était autorisé. Les compilateurs autorisent encore cette incohérence pour préserver la compatibilité descendante avec d'anciens codes sources, mais s'ils sont rigoureux ils doivent produire un message d'avertissement.

90 Avec les caractères accentués, ce décompte est plus subtil ; nous ne traitons pas ce cas ici.

Remarquez bien sur cet exemple les différences entre la fonction `strlen()` et l'opérateur `sizeof` : ce sont des sources de confusion fréquentes. Sur `str1` nous constatons en particulier que l'initialisation du tableau par la chaîne littérale ajoute implicitement le zéro terminal : sa taille fait un octet de plus que la longueur de la chaîne. Toutefois, comme constaté sur `str2`, rien n'empêche d'avoir un tableau plus grand que ce qui est nécessaire pour la chaîne. Sur `str3` nous constatons que ***l'opérateur `sizeof` appliqué à un pointeur donne la taille du pointeur lui-même et non la taille de ce qui est pointé***[91]. Cela se remarque aussi sur `str2+7` qui est considéré par l'opérateur `sizeof` comme un pointeur alors que `str2` est considéré comme le tableau dans sa globalité. Les cas de cet exemple mettent en évidence le fait que ***la fonction `strlen()` est un procédé dynamique qui compte un à un les caractères découverts à l'exécution alors que l'opérateur `sizeof` fournit une valeur déjà connue du compilateur par l'analyse préalable du type concerné***. Notez enfin que le type du résultat de la fonction `strlen()` est `size_t`, tout comme le résultat de l'opérateur `sizeof`[92], et nécessite donc un *cast* pour être considéré comme un simple entier.

5.3.2. Comparaison de chaînes

Puisque les chaînes de caractères ne sont pas des types de base, il n'existe aucun opérateur du langage sachant directement comparer leurs contenus (comme c'est le cas pour des entiers ou des réels par exemple). Cependant, ***puisque les chaînes sont manipulées par des variables de type pointeur ou tableau, la comparaison de ces variables est autorisée et se contente de comparer les adresses et non les contenus, ce qui est extrêmement trompeur !*** Il s'agit en effet d'arithmétique des pointeurs (chapitre 3).

```
char str1[]="same string";
const char *str2="same string";
char *str3=str1;
if(str1!=str2) { printf("str1!=str2\n"); } // str1!=str2
if(str1==str3) { printf("str1==str3\n"); } // str1==str3
```

Ici `str1` et `str2` désignent bien des contenus identiques mais qui sont stockés à des adresses différentes. De toute évidence `str3` est égale à `str1` puisqu'elle a été initialisée explicitement pour désigner la même adresse (celle du premier élément de `str1`).

La comparaison des contenus des chaînes est confiée à la fonction standard `strcmp()`.
(http://en.cppreference.com/w/c/string/byte/strcmp)
Elle n'effectue pas qu'un simple test d'équivalence des contenus mais une comparaison lexicographique pour établir une relation d'ordre. Il s'agit de comparer un à un les caractères en même position dans chacune des deux chaînes en partant du début. Si le code ASCII de l'un précède l'autre, alors la valeur de sa chaîne est immédiatement considérée inférieure à celle de l'autre. Tant que les caractères sont identiques, la comparaison se poursuit. Si les comparaisons sont identiques jusqu'au zéro terminal des deux chaînes, c'est quelles ont des contenus identiques. Si l'une d'elles est terminée avant l'autre, sa valeur est considérée comme inférieure à celle de l'autre. En résumé : pensez simplement à la façon dont sont ordonnés les mots dans un dictionnaire.

La fonction `strcmp()` indique par un entier le résultat de la comparaison des deux chaînes qu'elle reçoit en paramètres. Cet entier vaut zéro si les chaînes sont considérées identiques, il est négatif si la première est considérée inférieure à la seconde ou positif dans le cas contraire.

```
char str1[]="shorter", str2[]="same string";
const char *str3="same string", *str4="same string but longer";
printf("%d %d %d\n",strcmp(str1,str2),strcmp(str2,str3),strcmp(str3,str4)); // 7 0 -32
if(!strcmp(str2,str3)) { printf("str2 = str3\n"); }                          // str2 = str3
```

Remarquez que `str1` est plus courte que `str2` alors que leur comparaison désigne `str1` comme supérieure à `str2` (résultat positif). Le résultat nul de la comparaison de `str2` et `str3`

91 La taille d'un pointeur fait huit octets sur cet exemple ; il s'agit du cas particulier d'un système *64-bit*.

92 La justification de ce type a été donnée dans l'exercice pratique sur les types variés (chapitre 4).

indique que ces deux chaînes ont un contenu identique ; elles sont pourtant stockées à des emplacements distincts (voir l'exemple précédent). Les chaînes *str3* et *str4* sont identiques jusqu'à la fin de la première ; elle est donc inférieure (résultat négatif) à la seconde. La documentation de la fonction *strcmp()* précise que la valeur exacte du résultat importe peu : seuls le signe et la nullité ont du sens. La dernière comparaison utilisée comme condition d'une alternative est une construction très courante.

La fonction *strncmp()* est une variante de la précédente qui limite la comparaison à la longueur maximale indiquée en paramètre ; elle sert notamment à déterminer si une chaîne commence par une autre.

(http://en.cppreference.com/w/c/string/byte/strncmp)

5.3.3. Recherche dans une chaîne

Au delà de la comparaison, il peut être nécessaire de rechercher des informations dans une chaîne. Il s'agit essentiellement de retrouver un caractère particulier ou une sous-chaîne.

Les fonctions standards dédiées à ce propos sont *strchr()* et *strrchr()* pour rechercher un caractère respectivement à partir du début ou de la fin, et *strstr()* pour rechercher une sous-chaîne à partir du début.

(http://en.cppreference.com/w/c/string/byte/strchr)

(http://en.cppreference.com/w/c/string/byte/strrchr)

(http://en.cppreference.com/w/c/string/byte/strstr)

```
char str[]="demonstrate", *p;
p=strchr(str,'e');   if(p) { *p='E'; } // changer le premier 'e'
p=strrchr(str,'t');  if(p) { *p='T'; } // changer le dernier 't'
p=strstr(str,"str"); if(p) { *p='S'; } // changer le 's' de "str"
p=strchr(str,'s');   if(!p) { printf("no '%c' in <%s>\n",'s',str); } // no 's' in <dEmonStraTe>
```

Elles renvoient toutes les trois un pointeur sur l'élément recherché à l'intérieur de la chaîne inspectée, ou un pointeur nul s'il n'a pas été trouvé.

5.3.4. Recopie d'une chaîne

Puisque les chaînes de caractères ne sont pas des types de base, il n'existe aucun opérateur du langage sachant directement en recopier le contenu ; comme vu au chapitre 3, les tableaux ne peuvent pas être affectés et ***l'affectation de pointeurs affecte l'adresse mais pas le contenu d'une chaîne***. Recopier une chaîne revient alors à énumérer les caractères d'une chaîne source pour les recopier un à un vers une chaîne de destination, jusqu'à rencontrer le zéro terminal qui est lui même recopié afin de terminer la chaîne de destination. Bien entendu, cette dernière doit être modifiable et être préalablement dimensionnée pour être au moins aussi grande que la source.

La fonction standard *strcpy()* est dédiée à ce propos.

(http://en.cppreference.com/w/c/string/byte/strcpy)

```
const char *str1="first string";
char str2[100];
strcpy(str2,str1);
printf("<%s>\n",str2); // <first string>
strcpy(str2+6,str1);
printf("<%s>\n",str2); // <first first string>
```

Le tableau *str2* a initialement un contenu indéterminé mais la fonction *strcpy()* vient renseigner ses premiers octets avec ceux de la chaîne désignée par *str1*. Si nous savons ce que nous faisons, la destination de la copie ne doit pas forcément coïncider avec le début du tableau. Il convient toutefois d'être très prudent car ni le compilateur ni la fonction *strcpy()*

ne sont en mesure de déterminer si l'espace réservé pour la destination est suffisant (ici nous avons surdimensionné *str2* en prévision).

La fonction *strncpy()* est une variante de la précédente qui limite la recopie au nombre maximal d'octets indiqué en paramètre. L'écriture du zéro terminal n'est pas systématique (c'est une source d'erreur bien connue !) ; consultez très attentivement la documentation avant de vous en servir.

(http://en.cppreference.com/w/c/string/byte/strncpy)

Une variante de la recopie consiste en la concaténation : il s'agit de recopier une chaîne à la suite d'une autre. La fonction standard *strcat()* se charge de cette opération.

(http://en.cppreference.com/w/c/string/byte/strcat)

```
char str1[100]="one way";
char str2[100]="another way";
strcat(str1," to concatenate");
strcpy(str2+strlen(str2)," to do the same");
printf("<%s><%s>\n",str1,str2); // <one way to concatenate><another way to do the same>
```

Cet exemple illustre le fait que la concaténation est très similaire à la recopie. La même prudence est de mise concernant l'espace réservé pour la chaîne de destination.

La fonction *strncat()* est une variante de la précédente qui limite la recopie au nombre maximal de caractères indiqué en paramètre. Contrairement à *strncpy()*, l'écriture du zéro terminal est systématique.

(http://en.cppreference.com/w/c/string/byte/strncat)

Il est très important de retenir que ***les fonctions de recopie ou de concaténation de chaînes ne doivent être utilisées qu'en s'assurant que la chaîne de destination utilise un espace de stockage suffisamment grand pour accueillir tous les caractères de l'opération et le zéro terminal***. Sans cette précaution, les octets excédentaires écraseront probablement d'autres données, ce qui conduira à un comportement indéterminé du programme difficilement identifiable au débogueur. L'exercice pratique du chapitre 6 présente l'allocation dynamique de mémoire qui peut être utilisée avantageusement pour réserver un tel espace à la demande selon les conditions d'exécution du programme.

5.4. La ligne de commande

Jusqu'à maintenant, nos programmes ne reçoivent aucune consigne quant à leur fonctionnement. Leur seul paramétrage repose sur un ensemble de valeurs inscrites dans le code source et nécessite une nouvelle fabrication du programme exécutable à chaque modification de l'une d'elles.

Nous sommes déjà familiers de programmes auxquels nous passons des consignes lorsque nous les invoquons : le compilateur reçoit des options et le nom du fichier à compiler, l'éditeur de texte reçoit le nom du fichier à éditer... Toutes ces informations sont passées sur la ligne de commande. Ainsi, si nous réalisons un programme exécutable nommé *prog*, la commande :

```
$ ./prog there are some options ↲
```

provoque l'exécution de notre programme avec une ligne de commande constituée de son nom et des quatre mots suivants. La ligne de commande est une séquence de mots représentés chacun par une chaîne de caractères ; il y en a cinq sur cet exemple.

Le code source d'un programme peut y accéder en utilisant une nouvelle formulation pour le prototype de la fonction principale *main()*.

(http://en.cppreference.com/w/c/language/main_function)

```
int
main(int argc,    // argument counter : le nombre de mots de la ligne de commande
     char **argv) // argument values : le tableau des mots de la ligne de commande
{ /* ... */ return 0; }
```

Comme pour toute autre fonction, les noms des paramètres peuvent être choisis librement. Toutefois, ***tous les programmeurs utilisent les noms*** *argc* ***et*** *argv* ***pour nommer les paramètres de la fonction*** *main()*. Cette habitude permet de les désigner sans la moindre ambiguïté. Les types de ces paramètres sont en revanche imposés ; néanmoins, il arrive que le second soit formulé *char *argv[]*. En effet, le chapitre 3 précise qu'un tableau passé en paramètre n'est rien d'autre qu'un pointeur : la notation à base de crochets est, dans ces conditions, strictement équivalente à celle utilisant une étoile. Le paramètre *argv* est donc un tableau dont chaque élément est une chaîne de caractères. Ce tableau contient *argc* éléments correspondant chacun à un mot de la ligne de commande et un dernier élément qui est un pointeur nul. Tout se passe donc comme si, dans le cas présent, la fonction *main()* était invoquée de cette façon :

```
char a0[]="./prog", a1[]="there", a2[]="are", a3[]="some", a4[]="options";
char *commandLine[]={ a0, a1, a2, a3, a4, NULL };
int result=main(5,commandLine);
```

Bien entendu, cela n'est qu'illustratif : nous ne pouvons rien supposer quant aux détails effectifs de l'invocation de la fonction *main()* par le système d'exploitation.

Voici un programme typique qui énumère les arguments de sa ligne de commande. L'affichage qu'il produit correspond à la ligne de commande évoquée précédemment.

```
#include <stdio.h>
int
main(int argc,
     char **argv)
{                                          // argv[0]=<./prog>
for(int i=0;i<argc;++i)                    // argv[1]=<there>
  { printf("argv[%d]=<%s>\n",i,argv[i]); }  // argv[2]=<are>
return 0;                                  // argv[3]=<some>
}                                          // argv[4]=<options>
```

Un usage plus utile de ces arguments consisterait à analyser leur contenu pour conditionner l'exécution de différentes parties d'un programme.

Il est notamment possible d'extraire des valeurs numériques de ces chaînes de caractères en utilisant les fonctions *atoi()* ou *atof()* déclarées dans le fichier d'en-tête standard *stdlib.h*.

(http://en.cppreference.com/w/c/string/byte/atoi)

(http://en.cppreference.com/w/c/string/byte/atof)

Toutefois, ces fonctions ne permettent pas de savoir si la chaîne analysée décrivait effectivement une valeur numérique. Il est préférable d'utiliser à cette fin les fonctionnalités de formatage présentées dans l'exercice pratique du chapitre 8.

5.5. Résumé

Ce cours s'est largement appuyé sur les tableaux et les pointeurs pour introduire la notion de chaîne de caractères. Il ne s'agit en effet aucunement d'un type de base du langage mais d'une construction à partir d'une séquence de *char*.

Ces derniers sont de petits entiers dont la correspondance avec des caractères repose sur la table ASCII. Le langage offre une facilité pour les saisir de manière littérale et la bibliothèque standard propose des fonctions traitant ces caractères par catégories (lettres, chiffres…).

Les chaînes de caractères sont terminées par un caractère nul. Le langage offre une facilité pour les saisir de manière littérale et la bibliothèque standard propose des fonctions permettant de les manipuler (recopie, comparaison…).

La ligne de commande saisie lors du lancement d'un programme est accessible à ce dernier sous la forme d'un tableau de chaînes de caractères transmis à la fonction `main()`.

Bien qu'il s'agisse d'un type élaboré, les chaînes de caractères à zéro terminal sont tellement omniprésentes dans les programmes en langage C que leur maîtrise est aussi indispensable que celle des types de base.

6. L03_Alloc : Allocation dynamique

Motivation :

Dans le cours sur les pointeurs et les tableaux (chapitre 3), il est plusieurs fois mentionné le fait que les notions présentées devaient être étendues en abordant l'allocation dynamique de mémoire ; en voici donc le complément sous la forme d'un exercice pratique.

Au delà de l'occasion qui est offerte de s'exercer à la manipulation des pointeurs et des tableaux, ce sujet propose de découvrir avec l'allocation dynamique une nouvelle classe de stockage des données qui peut être considérée comme étant à mi-chemin entre les classes de stockage automatique et statique. Nous verrons également que ce procédé offre l'avantage de choisir la quantité de donnée qui est utile à un problème sans la connaître *a priori* : elle dépend des circonstances de l'exécution du programme.

Consignes :

Le travail demandé ici est extrêmement guidé puisqu'il s'agit de provoquer des cas problématiques afin de discuter de leur cause et éventuellement d'envisager des solutions. Veilliez donc à suivre scrupuleusement chacune des étapes en vous assurant à chaque fois d'avoir bien observé et compris ce qui vous est décrit.

6.1. Mise en place du programme

Ouvrez un terminal et grâce à la ligne de commande du *shell*, placez vous dans le répertoire de travail choisi pour ce sujet. Saisissez par exemple :

```
$ mkdir S3PRC_L03_Alloc ↲
$ cd S3PRC_L03_Alloc ↲
```

Placez dans ce répertoire le fichier `GNUmakefile` générique mis à votre disposition.

(http://www.enib.fr/~harrouet/Data/Courses/GNUmakefile)

Nous suivrons en effet la démarche de programmation modulaire présentée au chapitre 2.

Nous utilisons comme prétexte un ensemble de fonctionnalités autour des nombres premiers. La fonction `isPrime()` reproduite juste après utilise un procédé simple pour effectuer le test de primalité d'un nombre entier. Elle s'appuie sur le fait que les nombres premiers commencent à deux et donc que les suivants ne peuvent pas être pairs, d'où l'énumération de deux en deux à partir de trois. Elle cherche alors à déterminer si le nombre à tester est divisible par un des entiers énumérés. Si après avoir atteint la racine carrée du nombre testé aucune division ne tombe juste, c'est que le nombre testé est premier. Toutefois, pour accélérer la recherche, si des nombres premiers sont déjà connus et transmis dans un tableau alors la recherche d'un diviseur entier commence par ceux-ci. Cette fonction `isPrime()` sera déclarée dans le fichier d'en-tête `primes.h` (une inclusion est nécessaire pour le type `bool`) et définie dans le fichier `primes.c` (par simple copier/coller du code suivant).

```
bool // tested is a prime number
isPrime(const int *primes,
        int primeCount,
        int tested)
{
if(tested==2) { return true; }
if((tested<2)||!(tested%2)) { return false; }
for(int i=0;i<primeCount;++i)
  {
  const int div=primes[i];
  if(div*div>tested) { return true; }
  if(!(tested%div))  { return false; }
  }
const int lastDiv=primeCount>1 ? primes[primeCount-1] : 3;
for(int div=lastDiv;div*div<=tested;div+=2)
  { if(!(tested%div)) { return false; } }
return true;
}
```

Le programme principal sera rédigé dans le fichier *prog03.c* et appellera la fonction *test_isPrimes()* suivante, qui sera réalisée dans ce même fichier :

```
void
test_isPrime(void)
{
printf("\n~~~~ test_isPrime() ~~~~\n");
int primes[]={ 2, 3, 5, 7, 11, 13, 17, 19 };
const int primeCount=(int)(sizeof(primes)/sizeof(primes[0]));
for(int i=0;i<100;++i)
  { if(isPrime(primes,primeCount,i)) { printf("%d ",i); } }
printf("\n");
}
```

Elle se contente de déterminer les nombres premiers qui sont inférieurs à cent. La rédaction de ces trois fichiers doit respecter les consignes usuelles (chapitre 2).

Fabriquez ce programme et exécutez-le afin de vérifier que tout est en place. En l'état, il ne fait qu'afficher le nom de la fonction *test_isPrime()* et les nombres premiers inférieurs à cent :

$ make ↲

$ ./prog03 ↲

Assurez vous de constater que la séquence obtenue est correcte.

(https://primes.utm.edu/lists/small/)

6.2. Limite de la pile d'exécution

Nous cherchons maintenant à remplir un tableau de nombres premiers. Vous devez pour cela réaliser une fonction *populatePrimes()* qui attend en paramètres un tableau d'entiers à remplir (accompagné de son nombre d'éléments) et ne renvoie aucun résultat. Elle sera déclarée dans le fichier d'en-tête *primes.h* et définie dans le fichier *primes.c*.

Il lui suffit d'appeler en boucle la fonction *nextPrime()* suivante pour obtenir la valeur de chaque nouvel élément du tableau en fonction de ceux qui le précèdent :

```
static // only for internal use in this module
int // next prime number (after last of primes)
nextPrime(const int *primes,
          int primeCount)
{
if(primeCount<2) { return primeCount+2; } // start with 2 and 3
int tested=primes[primeCount-1];
do { tested+=2; } while(!isPrime(primes,primeCount,tested));
return tested;
}
```

Remarquez que la fonction *nextPrime()* est précédée du mot-clefs *static*[93] ; cela signifie qu'elle n'est visible que dans cette unité de compilation. En effet, cette fonction ne nous sera utile que dans *populatePrimes()* et d'autres fonctions du même module mais n'a aucune raison d'être invoquée depuis les autres modules de l'application. Elle n'est donc pas déclarée dans le fichier d'en-tête et son qualificatif *static* indique à l'éditeur de liens que, si par hasard une autre fonction à le même nom dans un autre module, alors il faudra la considérer comme une fonction distincte[94]. Puisque cette fonction n'est pas déclarée, sa définition devra précéder celle de votre fonction *populatePrimes()*.

Une fonction *test_populatePrimes()* sera réalisée dans le fichier *prog03.c* et sera appelée par le programme principal. Cette nouvelle fonction de test doit commencer par afficher son nom, déclarer un tableau *primes* de *1000* (mille) entiers non initialisés et en déterminer le nombre d'éléments *primeCount* comme précédemment. Votre fonction *populatePrimes()* servira à initialiser ce tableau ; il ne restera qu'à en visualiser le contenu avec la fonction suivante (simplement définie dans le même fichier) :

```
void
displayPrimes(const int *primes,
              int primeCount,
              int limit)
{
if(limit>0)
  {
  for(int i=0;i<limit;++i)
    { printf("%d%c",primes[i],(i%10)==9 ? '\n' : '\t'); }
  if(primeCount>limit) { printf("...\n"); }
  else if(limit%10)    { printf("\n"); }
  }
printf("last of %d primes: %d\n",
       primeCount,primeCount ? primes[primeCount-1] : 0);
}
```

Cette fonction affiche le contenu du tableau jusqu'à une certaine limite (pour éviter d'encombrer le terminal) ; vous vous contenterez ici des cent premiers éléments.

Fabriquez à nouveau ce programme et exécutez-le afin de vérifier qu'il produit bien la séquence de valeurs attendue (le millième nombre premier est *7919*). Reprenez ensuite cette expérience avec *100000* (cent-mille) éléments puis *1000000* (un-million), *2000000* (deux-millions), *3000000* (trois-millions), etc. Au delà du temps de calcul qui devient de plus en plus long, l'augmentation de la taille du tableau finira par provoquer un plantage du programme !

La zone mémoire utilisée par les appels de fonctions est la pile d'exécution. Elle contient notamment les paramètres et les variables de classe de stockage automatique de chaque fonction appelée. Le tableau que nous manipulons est bien une variable automatique, il est donc placé dans la pile d'exécution. La taille de cette dernière peut être très variable d'une

93 L'usage du mot-clef *static* sur une fonction n'a rien à voir avec les variables de classe de stockage statique !

94 Le *symbole* de cette fonction n'est pas *exporté* (selon le vocabulaire technique de l'édition de liens).

plateforme d'exécution à une autre (de quelques kilo-octets à quelques méga-octets) mais dans tous les cas elle est limitée. Par conséquent, ***il n'est pas raisonnable de placer dans la pile d'exécution des données exagérément volumineuses***. Cela implique qu'en pratique nous nous interdisons la déclaration de tableaux ayant plus que quelques dizaines voire quelques centaines d'éléments[95].

6.3. Allocation dynamique de mémoire

Lorsqu'il est question de manipuler un volume de données conséquent, il est nécessaire de recourir à une nouvelle classe de stockage : l'***allocation dynamique*** (vocabulaire). Il s'agit de réclamer au système d'exploitation un bloc d'octets d'une taille choisie afin que le programme en fasse l'usage qui lui convient. La zone de l'espace d'adressage dans laquelle le système d'exploitation puise pour fournir ces blocs est dénommée ***le tas*** (***heap***, vocabulaire).

Contrairement aux variables de classe de stockage automatique, un tel bloc de données reste accessible au programme, quelles que soient les entrées et les sorties dans les fonctions du programme, sans limitation de durée. Toutefois, si le programme estime qu'à partir d'un instant il n'a plus besoin d'un bloc de données, il doit le ***libérer*** (vocabulaire) explicitement afin que celui ci redevienne disponible pour un autre usage dans le système d'exploitation. L'oubli d'une telle libération est dénommée une ***fuite mémoire*** (vocabulaire). Si un programme produit de manière répétée des fuites mémoire, cette situation peut conduire à un extrême ralentissement de tout le système d'exploitation. Bien entendu, les blocs de données alloués par un programme seront automatiquement libérés lorsque ce dernier se terminera[96].

6.3.1. Prise en main

Une nouvelle fonction `test_allocatePrimes()` attendant un paramètre `primeCount` de type entier sera réalisée dans le fichier `prog03.c` et sera appelée par le programme principal en utilisant `1000` (mille) comme paramètre effectif. Le corps de cette fonction sera assez semblable à celui de la fonction `test_populatePrimes()`. Elle doit commencer par afficher son nom et son paramètre puis remplacer la déclaration du tableau d'entiers `primes` par un pointeur sur des entiers. Ce pointeur doit être initialisé par le résultat de l'appel à la fonction standard `malloc()`.

Pour ceci il vous faudra inclure le fichier d'en-tête précisé dans la documentation.

(http://en.cppreference.com/w/c/memory/malloc)

Le paramètre de `malloc()` représente la taille en octets du bloc de données réclamé. Ici, ce bloc devra contenir `primeCount` éléments de type `int` ; à vous d'en déduire sa taille en octets[97]. Puisqu'elle donne accès à une portion de la mémoire, cette fonction renvoie une adresse. Seulement, ses concepteurs n'ont aucune connaissance préalable de l'usage qui en sera fait ; il s'agit certes d'une adresse, mais celle d'une donnée de quel type ? C'est pour cette raison que le type de retour est `void *` : un pointeur vers un type indéterminé. Il vous faudra donc utiliser un *cast* pour convertir le type de l'adresse renvoyée vers le type de `primes`[98].

Désormais, le bloc de données désigné par le pointeur `primes` peut être considéré comme un tableau d'entiers obtenu dynamiquement. Il peut être utilisé comme l'était le tableau de la fonction `test_populatePrimes()` : nous le remplissons avec `populatePrimes()` et en affichons le contenu avec `displayPrimes()`. Après que l'affichage ait eu lieu, cette fonction simpliste n'a plus besoin du bloc de données obtenu par allocation dynamique ; ils nous faut alors le libérer avec la fonction standard `free()`.

La documentation de cette dernière nous indique qu'elle est déclarée dans le fichier d'en-tête déjà inclus pour la fonction d'allocation.

(http://en.cppreference.com/w/c/memory/free)

Son paramètre représente l'adresse du bloc mémoire dont nous n'avons plus l'usage. Il n'est pas question de fournir ici une adresse quelconque : cela produirait probablement un plantage. Au contraire, ***les seules adresses qui soient autorisées pour un appel à*** `free()` ***sont exactement celles qui ont été obtenues par les fonctions d'allocation dynamique***

95 Il en est tout autrement des tableaux ayant une classe de stockage statique (mais nous évitons d'y avoir recours).

96 Un programme qui produit des fuites mémoire finira probablement par un plantage, ce qui libérera sa mémoire.

97 Un compilateur rigoureux réclamera la conversion de la valeur de `primeCount` vers le type `size_t`.

98 Il est indiqué en 3.4.4 que les conversions de pointeurs sont rarement utiles ; voici une de ces rares exceptions.

(*malloc()* ici). Bien entendu, la libération d'un même bloc mémoire ne peut avoir lieu qu'une seule fois. Nous fournirons alors le pointeur *primes* comme paramètre à la fonction *free()*. Remarquez que, bien que le type attendu soit *void **, il n'est pas nécessaire d'utiliser un *cast*. En effet, l'adresse d'une donnée de type spécifique peut être vue comme une adresse quelconque, alors que l'inverse nécessite l'explicitation du type attendu.

Fabriquez à nouveau ce programme et exécutez-le afin de vérifier qu'il produit toujours la séquence de valeurs attendue (le millième nombre premier est *7919*). Reprenez ensuite cette expérience avec *100000* (cent-mille) éléments puis *1000000* (un-million), *2000000* (deux-millions), *3000000* (trois-millions), etc. Bien que le temps de calcul devienne de plus en plus long (toujours plus de nombres premiers à déterminer), le tableau d'entiers obtenu dynamiquement semble pouvoir atteindre une taille quelconque.

6.3.2. Digression : contrôle de l'échec de l'allocation

À lire la documentation de la fonction *malloc()*, nous en déduisons qu'il est prudent de contrôler la valeur du pointeur obtenu. Il est précisément annoncé que cette fonction renvoie un pointeur nul lorsque l'allocation échoue.

(http://en.cppreference.com/w/c/memory/malloc)

La première interrogation que la détection d'un tel événement soulève concerne le comportement à adopter lorsqu'il survient. S'il n'est effectivement pas possible d'obtenir du système d'exploitation le bloc mémoire dont le programme à besoin, que peut on faire d'autre que de terminer brutalement ce programme ? Tout action plus élaborée, comme afficher un message ou le consigner dans un journal d'activité, n'offre aucune garantie quant à son accomplissement puisqu'elle n'obtiendra peut-être pas la quantité de mémoire nécessaire à son propre fonctionnement ! La fonction standard *abort()*, sans paramètre ni résultat, déclarée dans le même fichier d'en-tête que les fonctions d'allocation, provoque à cet effet une terminaison anormale du programme qui peut être facilement détectée dans un débogueur[99].

(http://en.cppreference.com/w/c/program/abort)

```
int *primes=(int *)malloc((size_t)primeCount*sizeof(int));
if(!primes) { abort(); }
```

La terminaison du programme permettra certainement au système d'exploitation de regagner une quantité de mémoire qui lui faisait défaut ; cela sera d'autant plus significatif dans le cas où ce programme serait responsable d'une fuite mémoire causant l'épuisement constaté.

Il est cependant légitime de s'interroger sur le réel bénéfice d'une telle vérification. Si nous n'avions pas contrôlé la nullité du pointeur obtenu, et poursuivi le traitement de manière optimiste, alors à la toute première tentative de déréférencement de ce pointeur nous aurions provoqué un plantage du programme à cause d'un accès à une zone mémoire non autorisée (l'adresse nulle ici en l'occurrence) : une ***erreur de segmentation*** (vocabulaire). Dans les faits, cela provoque une terminaison du programme tout aussi anormale qu'avec l'utilisation de *abort()* ! C'est pour cette raison qu'il existe des partisans des deux stratégies suivantes : détecter le pointeur nul et terminer explicitement avec *abort()*, ou bien ne pas alourdir l'écriture du code avec un contrôle inutile et laisser le programme se terminer tout seul.

Pour compliquer la situation, il faut savoir que la plupart des systèmes d'exploitation distinguent la mémoire virtuelle de la mémoire physique[100]. Pour des raisons liées à l'optimisation des systèmes d'exploitation, lorsqu'une fonction d'allocation réclame un bloc de mémoire, il ne s'agit que de mémoire virtuelle. Ce n'est qu'à l'écriture dans cette zone mémoire que le système d'exploitation se préoccupe de la mémoire physique qu'il devra mobiliser pour rendre cette opération possible. Seulement, cette dernière vérification n'a lieu que bien après être sorti de la fonction *malloc()* (à l'utilisation du pointeur obtenu), et en cas d'échec de la mobilisation de la mémoire physique, une erreur de segmentation provoquera la

99 Convenons-en : si la mémoire vient à manquer, le débogueur ne pourra pas plus fonctionner que le programme !

100 Ces notions seront expliquées et étudiées dans les modules S7-CRS et S7-SEN de l'ENIB.

terminaison anormale du programme. Cela fait que dans la pratique il est quasiment impossible que `malloc()` renvoie un pointeur nul[101] : en cas d'épuisement de la mémoire physique le programme se plantera de toutes façons au déréférencement du pointeur.

Néanmoins, il peut arriver que sur quelques systèmes embarqués, équipés d'un système d'exploitation rudimentaire, la gestion de la mémoire soit bien plus simple et autorise un accès direct à la mémoire physique[102]. Si suite à l'épuisement de cette dernière, la fonction d'allocation renvoie un pointeur nul, et que nous l'utilisons sans précaution, alors nous risquons d'écraser les données qui sont situées dans la partie basse de l'espace d'adressage et donc de perturber fortement le système dans sa globalité (pas uniquement notre programme).

C'est finalement dans l'éventualité de la réutilisation de notre code sur une telle plateforme d'exécution que ***nous nous imposons de vérifier systématiquement la nullité du résultat de la fonction d'allocation pour, le cas échéant, terminer explicitement et brutalement le programme par la fonction standard*** `abort()`.

6.3.3. Intégration au module

Maintenant que nous avons expérimenté l'allocation dynamique de mémoire dans une fonction de test, nous pouvons envisager d'en proposer l'usage dans le module consacré aux nombres premiers. Il suffit pour cela de déclarer, dans le fichier d'en-tête `primes.h`, et définir, dans le fichier `primes.c`, une fonction `allocatePrimes()` qui attend un paramètre `primeCount` de type entier et qui renvoie comme résultat un tableau de `primeCount` entiers alloué dynamiquement et initialisé par la fonction `populatePrimes()`. Il est tout à fait correct ici de lui faire renvoyer un pointeur puisque les données situées à l'adresse indiquée resteront accessibles après la sortie de la fonction.

Modifiez alors la fonction `test_allocatePrimes()` de `prog03.c` pour qu'elle remplace ses invocations de `malloc()` et `populatePrimes()` par celle de la fonction `allocatePrimes()`. La libération de la mémoire reste toutefois à sa charge car elle seule sait quand le tableau d'entier obtenu dynamiquement n'est plus utile au programme. Fabriquez à nouveau ce programme et reprenez l'expérimentation précédente afin de vous assurer du fait que son comportement est identique.

6.4. Ré-allocation de la mémoire dynamique

Imaginons maintenant que nous ayons besoin de connaître toujours plus de nombres premier au cours de l'exécution du programme. Si la fonction `allocatePrimes()` nous en a déjà fourni et que plus tard nous en voulons plus, nous pourrions simplement libérer le premier tableau obtenu dynamiquement et invoquer une nouvelle fois `allocatePrimes()` avec un paramètre `primeCount` plus élevé. Seulement, cette opération nécessiterait un nouveau calcul des nombres premiers que nous connaissions déjà, ce qui représente une perte de temps.

Pour répondre à ce nouveau besoin, il nous faut réaliser une fonction `extendPrimes()` dont le rôle consiste à étendre un tableau de nombres premiers obtenu dynamiquement. Un tel tableau est désigné par un pointeur et un entier qu'il faudra modifier ; cette fonction reçoit donc en premier paramètre l'adresse d'un pointeur sur entiers nommée `inout_primes`, et en deuxième paramètre l'adresse d'un entier nommée `inout_primeCount`. Son troisième paramètre nommé `nextPrimeCount` est un simple entier indiquant le prochain nombre d'éléments de ce tableau. Elle doit être déclarée dans le fichier d'en-tête `primes.h` et définie dans le fichier `primes.c`.

Cette fonction commencera par obtenir dans des variables `primes` et `primeCount` une copie de ce qui est désigné par les paramètres `inout_primes` et `inout_primeCount`. Il est question d'obtenir un tableau avec un nouveau nombre d'éléments ; vous devrez donc obtenir dans une

101 Ceci signifierait l'épuisement de la mémoire virtuelle, mais l'espace d'adressage virtuel est généralement gigantesque devant la quantité de mémoire physiquement disponible.

102 Cette situation est rare cependant ; bon nombre de systèmes embarqués disposent désormais d'un système d'exploitation dont les fondements sont identiques à ceux des plateformes d'exécution usuelles.

variable *tmp* l'adresse de ce nouveau tableau d'entiers alloué dynamiquement. Les nombres premiers qui constituent le début de *tmp* sont identiques à ceux de *primes* ; il suffit de les recopier un à un. Le nombre de ces recopies est le minimum entre *primeCount* et *nextPrimeCount* (au cas où nous demanderions à réduire le tableau de nombres premiers). Désormais le bloc de données désigné par *primes* peut être libéré[103], et *tmp* peut être affecté à cette même variable ; *primes* désigne dorénavant un tableau ayant la taille requise et dont les premiers éléments sont les nombres premiers déjà connus. Il ne reste plus qu'à réaliser une boucle qui initialise par un appel à *nextPrime()* les élément situés entre *primeCount* et *nextPrimeCount*. Les paramètres *inout_primes* et *inout_primeCount* doivent enfin être utilisés pour donner aux variables qu'ils désignent les nouvelles valeurs de *primes* et *nextPrimeCount*.

Une nouvelle fonction *test_extendPrimes()* attendant deux paramètres *extendStep* et *extendCount* de type entier doit être réalisée dans le fichier *prog03.c* et appelée par le programme principal en utilisant *100000* (cent-mille) et *10* comme paramètres effectifs. Elle doit commencer par afficher son nom et ses paramètres puis déclarer un pointeur d'entiers *primes* et un entier *primeCount* ; ces deux variables doivent être initialement nulles. Une boucle de *extendCount* itérations sert à invoquer *extendPrimes()*. À chaque invocation *primes* et *primeCount* sont modifiés afin d'obtenir toujours plus de nombre premiers (par pas de *extendStep*) ; nous les afficherons avec *displayPrimes()*. À la fin de cette boucle, le dernier tableau obtenu dynamiquement doit être libéré. Fabriquez à nouveau ce programme et exécutez-le afin de vérifier qu'il produit toujours la séquence de valeurs attendue (le millionième nombre premier est *15485863*).

Le redimensionnement du tableau obtenu dynamiquement reposait sur une boucle de recopie des anciennes données vers le nouveau tableau. La fonction standard *memcpy()* est très couramment utilisée pour la copie d'un grand volume de données.

(http://en.cppreference.com/w/c/string/byte/memcpy)

Elle reçoit en paramètres les adresses des zones mémoire destination et source, ainsi que le nombre d'octets à recopier. Après avoir inclus le fichier d'en-tête précisé dans la documentation, remplacez la boucle de recopie de la fonction *extendPrimes()* par une invocation de *memcpy()*. Fabriquez à nouveau ce programme et reprenez l'expérimentation précédente afin de vous assurer du fait que son comportement est identique.

Finalement, la démarche qui consiste à allouer un nouvel espace mémoire, y recopier les données d'un précédent et libérer ce dernier correspond à un besoin assez courant. C'est pour cette raison qu'il existe la fonction standard de ré-allocation *realloc()*.

(http://en.cppreference.com/w/c/memory/realloc)

Elle reçoit en paramètres l'ancien pointeur et la taille en octets désirée pour le nouvel espace afin d'effectuer l'équivalent des trois étapes évoquées ici et de renvoyer la nouvelle adresse obtenue. Comme pour la fonction *free()*, ***il n'est pas question de fournir à la fonction realloc() autre chose qu'exactement une adresse préalablement obtenue par allocation dynamique*** (*malloc()* ou *realloc()* ici). Au-delà de l'économie d'écriture, elle offre un autre avantage bien plus intéressant : si l'ancien espace alloué est immédiatement suivi d'une zone de mémoire inutilisée qui est suffisamment grande, il est simplement redimensionné sur place sans nécessiter la moindre recopie (son adresse de début reste donc inchangée). Dans la pratique il n'est pas rare qu'il en soit ainsi, ce qui permet d'économiser assez souvent une longue opération de recopie. Remplacez alors, dans la fonction *extendPrimes()*, les trois étapes évoquées ici par un appel à *realloc()* en prenant soin de réaffecter son résultat dans *primes*. Puisqu'il s'agit d'une fonction d'allocation dynamique de mémoire, nous nous imposons, comme dans le cas de *malloc()*, de tester la nullité de son résultat pour invoquer *abort()* le cas échéant. Après cette ré-allocation il ne reste plus que la boucle pour déterminer les nombres premiers qui n'étaient pas encore connus et la mise à jour des variables désignées par les paramètres. Fabriquez à nouveau ce programme et reprenez l'expérimentation précédente afin de vous assurer du fait que son comportement est identique.

103 La documentation de *free()* précise que si le pointeur est initialement nul, sa libération ne pose aucun problème.

6.5. Allocation spéculative

Tous les exemples précédents avaient en commun le fait que nous connaissions exactement à l'avance le nombre d'entiers nécessaires au stockage des informations utiles. Il nous suffisait de réserver l'espace nécessaire (dans la pile ou dynamiquement) avant d'initialiser chacune des valeurs. Toutefois, il n'en est pas toujours ainsi : pour certains traitements il n'est pas aisé de déterminer à l'avance la quantité de données qui devront être mémorisées. C'est le cas par exemple si nous recherchons les nombres premiers jusqu'à un entier choisi : nous ne savons pas *a priori* combien de nombres premiers existent jusqu'à cette limite.

Nous pourrions envisager d'étendre le tableau d'un seul élément à la fois jusqu'à atteindre ou dépasser la limite recherchée mais ces multiples ré-allocations seraient largement sous-optimales en temps d'exécution. La solution alternative consiste à procéder de manière spéculative en allouant une quantité jugée probablement satisfaisante pour les prochains nombres premiers rencontrés et à ré-allouer de la même façon lorsque cette quantité n'est plus suffisante. Seulement, il n'est pas aisé d'estimer un pas de progression convenable pour ces multiples ré-allocations. Si nous choisissons un petit incrément (trente-deux éléments par exemple) mais que nous devons rechercher beaucoup de nombres premiers (jusqu'à dix-millions par exemple), il faudra recourir à énormément de ré-allocations. Or, ***les opérations d'allocation de mémoire sont très coûteuses en temps d'exécution ; il faut éviter de les invoquer de manière incessante***. À l'opposé, si l'incrément est très grand (un-million d'éléments par exemple) alors que nous ne recherchons que quelques nombres premiers (jusqu'à mille par exemple), nous consommerons beaucoup de mémoire inutilement[104].

Une démarche courante consiste à utiliser une progression géométrique de l'incrément afin qu'il soit de plus en plus grand. Il est généralement inutile de partir d'une quantité ridiculement petite ; nous pouvons fixer le premier incrément à trente-deux éléments par exemple. Une progression simple consiste à doubler cette quantité à chaque fois qu'une ré-allocation est nécessaire. Cependant les incréments risquent de devenir démesurément grands ; il convient donc de se fixer une limite, `1024*1024` éléments par exemple, au-delà de laquelle l'incrément ne sera plus doublé.

Pour expérimenter cette démarche, nous réalisons une fonction `allocatePrimesUntil()` dont le rôle consiste à allouer dynamiquement un tableau contenant les nombres premiers jusqu'à une limite choisie. Un tel tableau est désigné par un pointeur et un entier qu'il faudra renseigner ; cette fonction reçoit donc en premier paramètre l'adresse d'un pointeur sur entiers nommée `out_primes`, et en deuxième paramètre l'adresse d'un entier nommée `out_primeCount`. Son troisième paramètre nommé `lastTested` est un simple entier indiquant le dernier nombre pour lequel nous cherchons à déterminer s'il est premier. Elle doit être déclarée dans le fichier d'en-tête `primes.h` et définie dans le fichier `primes.c`.

Cette fonction repose sur trois variables initialement nulles : un pointeur sur entiers `primes` et deux entiers `capacity` et `primeCount`. Une boucle `for` sert à incrémenter `primeCount` mais n'exprime pas de condition puisque nous ne savons pas à l'avance la valeur que ce compteur atteindra. À chaque itération, un appel à la fonction `nextPrime()` fournit le prochain nombre premier `prime` ; s'il est strictement supérieur à `lastTested`, il faut quitter la boucle. Si `primeCount` a atteint `capacity`, cela signifie que nous n'avons pas la place pour mémoriser `prime` : il faut dans ce cas procéder à une ré-allocation de `primes`. Déterminez alors la nouvelle valeur de `capacity` en respectant la progression géométrique bornée décrite plus haut. Le tableau désigné par `primes` doit être ré-alloué selon cette nouvelle capacité[105]. Qu'une ré-allocation ait été nécessaire ou non, `prime` peut maintenant être mémorisé dans `primes`. Les paramètres `out_primes` et `out_primeCount` doivent enfin être utilisés pour donner aux variables qu'ils désignent les valeurs de `primes` et `primeCount`.

Une nouvelle fonction `test_allocatePrimesUntil()` attendant un paramètre `lastTested` de type entier doit être réalisée dans le fichier `prog03.c` et appelée par le programme principal en

104 Dans le cas présent (la détermination de nombres premiers) une heuristique pourrait probablement estimer directement une valeur approchée de l'incrément approprié mais ce n'est pas généralisable.

105 La documentation de `realloc()` précise que si le pointeur est initialement nul, il s'agit d'une simple allocation.

utilisant `1000000` (un-million) comme paramètre effectif. Elle doit commencer par afficher son nom et son paramètre puis déclarer un pointeur d'entiers `primes` et un entier `primeCount` qui seront renseignés par un appel à la fonction `allocatePrimesUntil()` pour obtenir les nombres entiers jusqu'à `lastTested`. Ils seront affichés avec `displayPrimes()` et le tableau obtenu dynamiquement pourra être libéré. Fabriquez à nouveau ce programme et exécutez-le afin de vérifier qu'il produit la séquence de valeurs attendue (il y a `78498` nombres premiers avant un-million ; le dernier d'entre eux est `999983`).

Utilisez maintenant le débogueur pour observer la progression des ré-allocations. Avancez dans la fonction `main()` avec `Next [F6]` jusqu'à l'abord de l'appel à `test_allocatePrimesUntil()`. Avec `Step [F5]`, entrez dans cette fonction et poursuivez avec `Next [F6]` jusqu'à l'abord de l'appel à `allocatePrimesUntil()`. Avec `Step [F5]`, entrez dans cette fonction et placez un point-d'arrêt (double-clic) sur l'appel à `realloc()`. Atteignez-le avec `Continue [F8]` et utilisez la commande `display` pour les variables `primes`, `primeCount` et `capacity`. Elles doivent valoir respectivement le pointeur nul, zéro et trente-deux. Progressez d'une ligne avec `Next [F6]` et observez les nouvelles valeurs affichées : primes a été renseigné. Reprenez l'exécution avec `Continue [F8]` : au point-d'arrêt, `primeCount` vaut trente-deux et `capacity` a doublé. Poursuivez en alternant `Next [F6]` et `Continue [F8]` pour observer l'évolution des variables avant et après chaque ré-allocation. Il est fort probable (mais non garanti) que le bloc de données désigné par `primes` ne soit jamais déplacé. Nous en déduisons qu'aucune recopie des anciennes valeurs n'a lieu lors des ré-allocations ce qui justifie pleinement notre choix de recourir l'usage de `realloc()` pour éviter la combinaison `malloc()`/`memcpy()`/`free()`.

À ce stade, vous devez être en mesure de réaliser une fonction `extentPrimesUntil()` qui repose sur les mêmes principes que `allocatePrimesUntil()` et `extendPrimes()` (la capacité initiale du tableau sera supposée correspondre au nombre de ces éléments). Remarquez que l'algorithme de cette nouvelle fonction est très similaire à celui `allocatePrimesUntil()` et que cette dernière peut être réécrite comme un simple appel à la nouvelle.

La fonction `test_extendPrimesUntil()` sera similaire à `test_extendPrimes()` et sera invoquée pour effectuer dix pas cherchant à atteindre le prochain million. Fabriquez à nouveau ce programme et exécutez-le afin de vérifier qu'il produit la séquence de valeurs attendue (il y a `664579` nombres premiers avant dix-millions ; le dernier d'entre-eux est `9999991`).

6.6. Résumé

Ici s'achève cette troisième séance pratique dans laquelle nous nous sommes familiarisés avec l'allocation dynamique de mémoire. Au delà de la présentation de quelques fonctionnalités standards, nous avons mis en évidence les besoins auxquels elles répondent ainsi que les règles d'usage et les précautions à observer.

Voici un récapitulatif des points à retenir et à maîtriser :

- les tableaux de classe d'allocation automatique ne doivent pas contenir beaucoup d'éléments (la pile d'exécution a une taille très limitée),
- l'allocation dynamique permet d'obtenir des blocs d'octets de taille choisie librement (même extrêmement grande),
- un tel bloc peut être vu comme une tableau obtenu dynamiquement contenant `ELEM_COUNT` d'éléments de type `TYPE`,
  `TYPE *pointeur=(TYPE *)malloc((size_t)ELEM_COUNT*sizeof(TYPE));`
- en cas d'échec de l'allocation il faut terminer brutalement le programme avec `abort()`,
- les blocs d'octets obtenus dynamiquement sont accessibles jusqu'à leur libération explicite par la fonction `free()`,
- la fonction `free()` n'est applicable qu'à l'adresse de début d'un bloc d'octets obtenu dynamiquement et ne doit libérer qu'une seule fois un même bloc,
- l'oubli de la libération de blocs d'octets obtenus dynamiquement alors qu'ils ne sont plus utiles dans le programme constitue une fuite mémoire et peut conduire à l'épuisement de la mémoire disponible dans le système,
- la fonction `realloc()` n'est applicable qu'à l'adresse de début d'un bloc d'octets obtenu dynamiquement et sert à changer la taille de ce bloc,

- un bloc ré-alloué peut changer d'emplacement, il faut réaffecter son pointeur, `pointeur=(TYPE *)realloc(pointeur,(size_t)ELEM_COUNT*sizeof(TYPE));`
- la ré-allocation conserve les données du bloc redimensionné en recourant éventuellement à une recopie,
- si la ré-allocation d'un bloc peut avoir lieu sur place, elle économise la recopie des données,
- l'allocation spéculative consiste à surestimer temporairement la taille du bloc à allouer et à le ré-allouer lorsque ce n'est plus suffisant,
- il faut éviter d'invoquer de manière incessante les opérations d'allocation de mémoire puisqu'elles sont en général coûteuses en temps d'exécution.

Pour vous entraîner, et en vue de passer l'épreuve pratique, veuillez réaliser un programme qui manipule des chaînes de caractères (découvertes dans le cours du chapitre 5) par allocation dynamique pour effectuer quelques traitements simples (recopie, redimensionnement...).

Voici le contenu des fichiers réalisés dans ce sujet, tels qu'ils doivent être si vous avez suivi scrupuleusement les recommandations qui vous ont été données à chaque étape.

```
//---- prog03.c ----
#include <stdio.h>
#include <stdlib.h>
#include "primes.h"

void
displayPrimes(const int *primes,
              int primeCount,
              int limit)
{
if(limit>0)
  {
  for(int i=0;i<limit;++i)
    { printf("%d%c",primes[i],(i%10)==9 ? '\n' : '\t'); }
  if(primeCount>limit) { printf("...\n"); }
  else if(limit%10)    { printf("\n"); }
  }
printf("last of %d primes: %d\n",
       primeCount,primeCount ? primes[primeCount-1] : 0);
}

void
test_isPrime(void)
{
printf("\n~~~~ test_isPrime() ~~~~\n");
int primes[]={ 2, 3, 5, 7, 11, 13, 17, 19 };
const int primeCount=(int)(sizeof(primes)/sizeof(primes[0]));
for(int i=0;i<100;++i)
  { if(isPrime(primes,primeCount,i)) { printf("%d ",i); } }
printf("\n");
}

void
test_populatePrimes(void)
{
printf("\n~~~~ test_populatePrimes() ~~~~\n");
int primes[1000];
const int primeCount=(int)(sizeof(primes)/sizeof(primes[0]));
populatePrimes(primes,primeCount);
displayPrimes(primes,primeCount,100);
}

void
test_allocatePrimes(int primeCount)
{
printf("\n~~~~ test_allocatePrimes(%d) ~~~~\n",primeCount);
int *primes=allocatePrimes(primeCount);
displayPrimes(primes,primeCount,0);
free(primes);
}


                                                    // ...(1/2)...
```

```
                                                                    // ...(2/2)...

void
test_extendPrimes(int extentStep,
                  int extentCount)
{
printf("\n~~~~ test_extendPrimes(%d,%d) ~~~~\n",extentStep,extentCount);
int primeCount=0;
int *primes=NULL;
for(int i=0;i<extentCount;++i)
  {
  extendPrimes(&primes,&primeCount,(i+1)*extentStep);
  displayPrimes(primes,primeCount,0);
  }
free(primes);
}

void
test_allocatePrimesUntil(int lastTested)
{
printf("\n~~~~ test_allocatePrimesUntil(%d) ~~~~\n",lastTested);
int primeCount;
int *primes=NULL;
allocatePrimesUntil(&primes,&primeCount,lastTested);
displayPrimes(primes,primeCount,0);
free(primes);
}

void
test_extendPrimesUntil(int extentStep,
                       int extentCount)
{
printf("\n~~~~ test_extendPrimesUntil(%d,%d) ~~~~\n",extentStep,extentCount);
int primeCount=0;
int *primes=NULL;
for(int i=0;i<extentCount;++i)
  {
  extendPrimesUntil(&primes,&primeCount,(i+1)*extentStep);
  displayPrimes(primes,primeCount,0);
  }
free(primes);
}

int
main(void)
{
test_isPrime();
test_populatePrimes();
test_allocatePrimes(100000);
test_extendPrimes(100000,10);
test_allocatePrimesUntil(1000000);
test_extendPrimesUntil(1000000,10);
return 0;
}
```

```
//---- primes.h ----
#ifndef PRIMES_H
#define PRIMES_H 1

#include <stdbool.h>

bool // tested is a prime number
isPrime(const int *primes,
        int primeCount,
        int tested);

void
populatePrimes(int *primes,
               int primeCount);

int * // allocated prime numbers
allocatePrimes(int primeCount);

void
extendPrimes(int **inout_primes,
             int *inout_primeCount,
             int nextPrimeCount);

void
allocatePrimesUntil(int **out_primes,
                    int *out_primeCount,
                    int lastTested);

void
extendPrimesUntil(int **inout_primes,
                  int *inout_primeCount,
                  int lastTested);

#endif // PRIMES_H
```

```
//---- primes.c ----
#include "primes.h"
#include <stdlib.h>
#include <string.h>

#define USE_REALLOC 1
#define USE_MEMCPY  1

bool // tested is a prime number
isPrime(const int *primes,
        int primeCount,
        int tested)
{
if(tested==2) { return true; }
if((tested<2)||!(tested%2)) { return false; }
for(int i=0;i<primeCount;++i)
  {
  const int div=primes[i];
  if(div*div>tested) { return true; }
  if(!(tested%div))  { return false; }                                    // ...(1/4)...
```

```
  }                                                                          // ...(2/4)...
const int lastDiv=primeCount>1 ? primes[primeCount-1] : 3;
for(int div=lastDiv;div*div<=tested;div+=2)
  { if(!(tested%div)) { return false; } }
return true;
}

static // only for internal use in this module
int // next prime number (after last of primes)
nextPrime(const int *primes,
          int primeCount)
{
if(primeCount<2) { return primeCount+2; } // start with 2 and 3
int tested=primes[primeCount-1];
do { tested+=2; } while(!isPrime(primes,primeCount,tested));
return tested;
}

void
populatePrimes(int *primes,
               int primeCount)
{
for(int i=0;i<primeCount;++i)
  { primes[i]=nextPrime(primes,i); }
}

int * // allocated prime numbers
allocatePrimes(int primeCount)
{
int *primes=(int *)malloc((size_t)primeCount*sizeof(int));
if(!primes) { abort(); }
populatePrimes(primes,primeCount);
return primes;
}

void
extendPrimes(int **inout_primes,
             int *inout_primeCount,
             int nextPrimeCount)
{
//-- load inout-parameters --
int *primes=*inout_primes;
const int primeCount=*inout_primeCount;
//-- perform work --
#if USE_REALLOC
  primes=(int *)realloc(primes,(size_t)nextPrimeCount*sizeof(int));
  if(!primes) { abort(); }
#else
  int *tmp=(int *)malloc((size_t)nextPrimeCount*sizeof(int));
  if(!tmp) { abort(); }
  const int limit=primeCount<nextPrimeCount ? primeCount : nextPrimeCount;
# if USE_MEMCPY
    memcpy(tmp,primes,(size_t)limit*sizeof(int));
# else
    for(int i=0;i<limit;++i)
      { tmp[i]=primes[i]; }
# endif                                                                      // ...(2/4)...
```

```
  free(primes);                                                                  // ...(3/4)...
  primes=tmp;
#endif
for(int i=primeCount;i<nextPrimeCount;++i)
  { primes[i]=nextPrime(primes,i); }
//-- store out-parameters --
*inout_primes=primes;
*inout_primeCount=nextPrimeCount;
}

#if 0 // first version of allocatePrimesUntil()

void
allocatePrimesUntil(int **out_primes,
                    int *out_primeCount,
                    int lastTested)
{
int *primes=NULL;
int capacity=0, primeCount;
for(primeCount=0;;++primeCount)
  {
  const int prime=nextPrime(primes,primeCount);
  if(prime>lastTested) { break; }
  if(primeCount==capacity)
    {
    const int minChunk=32, maxChunk=1024*1024;
    if(capacity<minChunk)      { capacity=minChunk; }
    else if(capacity<maxChunk) { capacity*=2; }
    else                       { capacity+=maxChunk; }
    primes=(int *)realloc(primes,(size_t)capacity*sizeof(int));
    if(!primes) { abort(); }
    }
  primes[primeCount]=prime;
  }
//-- store out-parameters --
*out_primes=primes;
*out_primeCount=primeCount;
}

#else // second version of allocatePrimesUntil()

void
allocatePrimesUntil(int **out_primes,
                    int *out_primeCount,
                    int lastTested)
{
int *primes=NULL;
int primeCount=0;
extendPrimesUntil(&primes,&primeCount,lastTested);
//-- store out-parameters --
*out_primes=primes;
*out_primeCount=primeCount;
}

#endif

                                                                                 // ...(3/4)...
```

```
                                                                                // ...(4/4)...

void
extendPrimesUntil(int **inout_primes,
                  int *inout_primeCount,
                  int lastTested)
{
//-- load inout-parameters --
int *primes=*inout_primes;
int capacity=*inout_primeCount;
//-- perform work --
int primeCount;
for(primeCount=capacity;;++primeCount)
  {
  const int prime=nextPrime(primes,primeCount);
  if(prime>lastTested) { break; }
  if(primeCount==capacity)
    {
    const int minChunk=32, maxChunk=1024*1024;
    if(capacity<minChunk)      { capacity=minChunk; }
    else if(capacity<maxChunk) { capacity*=2; }
    else                       { capacity+=maxChunk; }
    primes=(int *)realloc(primes,(size_t)capacity*sizeof(int));
    if(!primes) { abort(); }
    }
  primes[primeCount]=prime;
  }
//-- store out-parameters --
*inout_primes=primes;
*inout_primeCount=primeCount;
}
```

7. C04_Struct : Structures de données

Les seuls types que nous ayons manipulés jusqu'alors sont les types de base fournis par le langage. Il s'agit essentiellement de données numériques (entiers, réels, caractères...) et d'adresses (pointeurs, tableaux, chaînes...). Le mot-clef *typedef* (présenté au chapitre 1) permet d'inventer un nouveau nom de type mais celui-ci ne représente rien d'autre qu'un des types déjà existants. Les tableaux ont la particularité de permettre la désignation de plusieurs valeurs distinctes par une unique variable (et un indice entier), seulement les valeurs d'un même tableau ont toutes un type identique.

Nous verrons ici que le langage C nous offre la possibilité d'aller au-delà de ces quelques types en inventant de nouveaux types de données. Il s'agit d'agréger plusieurs types existants pour former une structure de données. Au delà des détails du langage, nous présenterons une démarche de mise en œuvre systématique de cette fonctionnalité.

7.1. Un nouveau type de données

Nous abordons cette nouvelle notion par la découverte des éléments du langage qui permettent de la manipuler.

7.1.1. Définition d'une structure

Une ***structure*** (vocabulaire) regroupe un ou plusieurs ***membres*** (ou champs, vocabulaire), c'est à dire un ensemble de variables qui sont constitutives d'une telle structure.

(http://en.cppreference.com/w/c/language/struct)

```
struct MyStruct        // définition de la structure nommée MyStruct
{
int i,j;               // une instance de la structure MyStruct est constituée de deux entiers
double d;              // et d'un réel
};
struct MyStruct v1,v2; // les variables v1 et v2 sont des instances de la structure MyStruct
```

Le mot-clef *struct* permet d'inventer un nouveau type : *struct MyStruct* ici. La définition de ce type revient à déclarer des variables de type quelconque dans la paire d'accolades associée (le point-virgule final est obligatoire)[106]. Ce nouveau type, comme n'importe quel autre, peut ensuite servir à déclarer des variables, des paramètres, le type de retour d'une fonction...

Généralement, une telle définition de structure est placée à l'extérieur de toute fonction, dans un fichier d'en-tête par exemple, mais rien n'empêche d'y avoir recours localement pour un usage très spécifique à une fonction.

Le nom du nouveau type ainsi introduit est composé de deux mots : le mot-clef *struct* et le nom que nous avons choisi de lui donner. Ceci alourdit l'écriture du code et beaucoup de programmeurs préfèrent désigner le type d'une structure par un mot unique grâce à l'utilisation du mot-clef *typedef*[107].

106 Il existe des variantes dans la syntaxe (déclaration prématurée, structures anonymes directement associées à des variables...) mais nous nous focalisons ici sur la forme la plus courante.

107 En langage C++ le seul nom de la structure suffit à désigner son type ; *typedef* ne serait pas nécessaire ici.

```
struct MyStruct { int i,j; double d; }; // définition du type struct MyStruct
typedef struct MyStruct MyType;         // le type nommé MyType équivaut à struct MyStruct
MyType v1,v2;                           // deux instances de struct MyStruct
```

Dans la pratique, toujours dans un soucis d'économie d'écriture, il est très courant de fusionner ces deux définitions de types (*struct* et *typedef*).

```
typedef struct { int i,j; double d; } MyType;               // une structure simple
typedef struct Node { int value; struct Node *next; } Node; // une structure auto-référencée
```

Remarquez que, dans cette forme, le nom qui suit le mot-clef *struct* est optionnel. Il est utile dans le cas d'une structure auto-référencée, comme *Node* ici, dont un membre est un pointeur vers ce même type : le compilateur ne connaît pas encore le nom attribué avec *typedef*, le type du membre doit donc utiliser le nom complet de la structure.

Ces nouveaux types étant utilisables comme n'importe quel autre, ils peuvent également servir de membre pour d'autres structures.

```
typedef struct { double x,y,z; } Vec3D;
typedef struct
{
double radius,mass;
Vec3D pos,vel;
} Particle;
Particle p1,p2;
```

Ici une première structure décrit un triplet de réels représentant un vecteur de l'espace tridimensionnel. Une particule est un type un peu plus élaboré qui contient, en plus de ses propriétés de rayon et de masse, deux tels vecteurs tridimensionnels pour représenter sa position et sa vitesse. Les variables *p1* et *p2* de cet exemple sont toutes les deux constituées de toutes ces informations. Nous constatons ici l'intérêt de cette notion : ***une structure permet de regrouper dans une même entité logique des données dont la signification serait confuse si elles étaient éparses***.

7.1.2. Accès aux membres d'une structure

Puisqu'une unique variable de type structure peut être constituée de plusieurs membres, le langage doit proposer une notation pour distinguer chacun d'eux. L'opérateur `.` (point) est dédié à ce propos.

(http://en.cppreference.com/w/c/language/operator_member_access)

```
Particle p;                                     // type Particle de l'exemple précédent
p.radius=0.5;                                   // modification d'un membre
p.vel.z=-0.8;                                   // modification d'un sous-membre
printf("radius=%g vel_z=%g\n",p.radius,p.vel.z); // radius=0.5 vel_z=-0.8
```

L'usage de cet opérateur est trivial : il désigne le membre dont le nom suit au sein de la variable dont le nom précède. Chaque membre ainsi désigné est alors vu comme n'importe quelle variable usuelle que l'on peut modifier, consulter, dont on peut prendre l'adresse... Remarquez que si un membre est lui-même une structure (*p.vel* ici) , un nouvel opérateur `.` (point) permet de désigner un membre de cette structure imbriquée.

Les structures étant des variables comme les autres il est tout à fait envisageable d'en relever l'adresse ou de les placer dans un tableau (alloué dynamiquement ou non). Ceci fait qu'elles peuvent être désignées par des pointeurs. Il est ainsi possible de déréférencer un tel pointeur pour évoquer la structure désignée, puis d'utiliser l'opérateur `.` (point) pour accéder à l'un de ses membres. L'opérateur `->` (flèche) permet toutefois de simplifier l'écriture.

```
typedef struct { int i,j; double d; } MyType;
MyType v1, *v2=&v1, v3[4];
v2->i=1; v2->j=10; v2->d=1.1; // équivaut à (*v2).i=1; (*v2).j=10; (*v2).d=1.1;
for(int i=0;i<4;++i)
  { v3[i].i=1+v2->i; v3[i].j=10+v2->j; v3[i].d=1.1+v2->d; v2=v3+i; }
printf("last: i=%d j=%d d=%g\n",v2->i,v2->j,v2->d);                  // last: i=5 j=50 d=5.5
```

Nous constatons ici que l'invocation de *`pointeur->membre`* n'est qu'une forme plus lisible pour la notation *`(*pointeur).membre`*. Remarquez que l'opérateur `.` (point) est plus prioritaire que l'opérateur de déréférencement ce qui imposerait l'usage des parenthèses si nous n'utilisions pas l'opérateur `->` (flèche).

7.1.3. Initialisation d'une structure

Dans les quelques exemples précédents, toutes nos variables de type structure étaient déclarées sans être initialisées. Elles avaient donc, comme tout autre type de variables, une valeur indéterminée à leur apparition. Il nous fallait alors en initialiser tous les membres un à un. Il existe toutefois une notation qui permet l'initialisation de tous les membres d'une variable de type structure lors de la déclaration ; celle-ci est très proche de ce que nous connaissons à propos de l'initialisation des tableaux (voir en 3.3.2).

(http://en.cppreference.com/w/c/language/struct_initialization)

```
typedef struct { int i,j; double d; } MyType;
MyType v1;                      // les membres ont des valeurs indéterminées
MyType v2={ 1, 10, 1.1 };       // les trois membres explicitement initialisés
MyType v3={ 1, 10 };            // les deux premiers membres spécifiés, le dernier à zéro
MyType v4[]={ { 2, 20, 2.2 },   // initialisation des trois structures d'un tableau
              { 3 }, { 4, 40} };
Particle v5={ 0.1, 4.5,           // membres radius et mass (type Particle de l'exemple en 7.1.1)
              { 0.0, 0.0, 5.0 } ,  // membre pos
              { 1.0, 0.0, 1.0 } }; // membre vel
```

Il s'agit en effet d'une liste d'initialisation qui énumère, dans une paire d'accolades et séparées par des virgules, les valeurs initiales dans le même ordre que les membres[108]. Tout comme en ce qui concerne les tableaux, si au moins un membre est spécifié, tous les autres sont implicitement initialisés à zéro. Les listes d'initialisation des tableaux et des structures peuvent être imbriquées (tableaux de structures, structures imbriquées, structures dont des membres sont des tableaux...). La liste d'initialisation d'une structure ne peut être utilisée qu'à la déclaration d'une variable et ne peut pas servir à une affectation ultérieure ni comme paramètre effectif d'un appel de fonction. Ceci est cohérent avec ce ce que nous connaissons à propos de la liste d'initialisation des tableaux.

7.1.4. Copie d'une structure

Les variables de type structure peuvent être directement recopiées comme le sont les types de base ; il s'agit là d'une différence fondamentale avec les tableaux. Ces recopies peuvent avoir lieu suite à l'usage de l'opérateur d'affectation, à l'occasion d'un passage de paramètre à une fonction, ou encore lors du retour d'un résultat par une fonction.

```
typedef struct { int i,j; double d; } MyType;
MyType doSomething(MyType v)           // passage par valeur, recopie du paramètre effectif
{ v.i*=2; v.j*=3, v.d*=4.0; return v; } // modification du paramètre formel et renvoi d'une copie

// ... dans une autre fonction ...
MyType v1={ 1, 10, 1.1 }, v2;
v2=doSomething(v1);               // passage par valeur, renvoi puis affectation
printf("v1: i=%d j=%d d=%g\n",v1.i,v1.j,v1.d); // v1: i=1 j=10 d=1.1
printf("v2: i=%d j=%d d=%g\n",v2.i,v2.j,v2.d); // v2: i=2 j=30 d=4.4
```

108 Une notation permettant de désigner les membres existe dans la version C99 mais n'est pas supportée en C++.

Nous constatons en particulier sur cet exemple que la variable *v1* servant de paramètre effectif à l'appel de la fonction n'est pas du tout modifiée malgré la modification du paramètre formel par cette dernière ; il s'agit donc bien d'un passage par valeur contrairement aux tableaux. Les multiples recopies effectuées transmettent bien les valeurs de tous les membres.

Pour aller encore plus loin sur les points communs et les différences de comportement entre la manipulation de structures et de tableaux, envisageons le cas où un membre d'une structure est un tableau.

```
typedef struct { double xyz[3]; } XYZ;
XYZ doSomething(XYZ v)
{ v.xyz[0]*=2.0; v.xyz[1]*=3.0; v.xyz[2]*=4.0; return v; }

// ... dans une autre fonction ...
XYZ v1={ { 1.1, 2.2, 3.3 } }, v2;
v2=doSomething(v1);                // passage par valeur, renvoi puis affectation
printf("v1: %g %g %g\n",v1.xyz[0],v1.xyz[1],v2.xyz[2]); // v1: 1.1 2.2 3.3
printf("v2: %g %g %g\n",v2.xyz[0],v2.xyz[1],v2.xyz[2]); // v2: 2.2 6.6 13.2
```

Ce nouvel exemple reprend le même scénario que le précédent. Nous constatons que, bien qu'un membre de la structure soit un tableau, la recopie des contenus des structures a bien lieu comme attendu. Nous en déduisons que si dans une application nous avons besoin de recopier des tableaux dont le nombre d'éléments est bien déterminé à la compilation, nous pouvons contourner la limitation imposée par le langage en inventant un type de structure contenant un tel tableau. Attention toutefois, ceci n'est vrai que pour les tableaux déclarés avec des crochets : ***lors de la recopie d'une structure, les membres qui sont des pointeurs sont recopiés comme de simples pointeurs, c'est à dire que les données qu'ils désignent ne sont pas recopiées*** (seule l'adresse qu'ils mémorisent l'est).

7.1.5. Disposition des membres d'une structure

Puisqu'une structure peut être constituée de multiples membres, une variable de ce type doit occuper en mémoire un ensemble d'octets suffisant à leur mémorisation. L'opérateur *sizeof*, découvert au chapitre 4, s'applique bien entendu aux types qui sont des structures.

```
typedef struct { char a; char b; char c; char d; int e; int  f; int g; int  h; } S1;
typedef struct { char a;                         int e; int  f; int g; int  h; } S2;
typedef struct { int  e; char a; int  f; char b; int g; char c; int h; char d; } S3;
printf("sizeof: char=%d\tint=%d\n",                      // sizeof: char=1  int=4
       (int)sizeof(char),(int)sizeof(int));
printf("sizeof: S1=%d\tS2=%d\tS3=%d\n",                  // sizeof: S1=20   S2=20   S3=32
       (int)sizeof(S1),(int)sizeof(S2),(int)sizeof(S3));
```

La norme impose qu'un *char* occupe un octet (un *byte* exactement) et sur la plateforme d'exécution ayant servi à cet exemple nous constatons qu'un *int* en occupe quatre. La structure *S1* qui est constituée de quatre *int* et quatre *char* occupe bien les vingt octets attendus, ce qui semble naturel. En revanche, la structure *S2* qui contient trois *char* de moins fait également la même taille que *S1*, ce qui est contre-intuitif. La structure *S3* contient, quant à elle, les mêmes membres que *S1* mais disposés dans un ordre différent, ce qui, étonnamment, augmente sa taille.

L'explication de ce phénomène tient dans la notion d'***alignement*** (vocabulaire) des données. Tout comme chaque type a une propriété de taille, il a une propriété d'alignement. Celle-ci traduit le fait qu'une donnée d'un certain type ne peut être située à une adresse quelconque. De manière générale, en ce qui concerne les types de base, une donnée d'un seul octet n'a aucune contrainte d'adresse, une donnée de deux octets doit être située à une adresse paire, une donnée de quatre octets doit être située à une adresse multiple de quatre, etc. Toutefois, ces contraintes ne sont pas toujours aussi systématiques et dépendent très fortement du

processeur qui exécutera le code[109]. Le compilateur connaît exactement les propriétés spécifiques à la plateforme d'exécution et choisit d'aligner les données en conséquence.

(http://en.cppreference.com/w/c/language/object)

```
int byteOffset(const void *addr1, const void *addr2) // l'écart en octets entre deux adresses
{ return (int)((const char *)addr2-(const char *)addr1); }

// ... dans une autre fonction ...
S1 v1; S2 v2; S3 v3; // les même structures S1, S2 et S3 que dans l'exemple précédent
printf("S1: a_to_b=%d a_to_e=%d e_to_f=%d\n",                    // S1: a_to_b=1 a_to_e=4 e_to_f=4
       byteOffset(&v1.a,&v1.b),byteOffset(&v1.a,&v1.e),
       byteOffset(&v1.e,&v1.f));
printf("S2: a_to_e=%d e_to_f=%d\n",                              // S2: a_to_e=4 e_to_f=4
       byteOffset(&v2.a,&v2.e),byteOffset(&v2.e,&v2.f));
printf("S3: e_to_a=%d a_to_f=%d f_to_b=%d\n",                    // S3: e_to_a=4 a_to_f=4 f_to_b=4
       byteOffset(&v3.e,&v3.a),byteOffset(&v3.a,&v3.f),
       byteOffset(&v3.f,&v3.b));
```

Nous constatons ici que les quatre `char` de la structure *S1* ne sont séparés que d'un octet et que les `int` le sont de quatre. Dans la structure *S2*, l'espace entre l'unique `char` et le premier `int` est toujours de quatre octets alors qu'un seul octet est utile. Ceci est dû au fait que le compilateur a décidé d'aligner le prochain `int` pour qu'il soit situé à une adresse multiple de quatre ; pour ceci il le fait précéder de trois octets de bourrage. Le cas de la structure *S3* est exagérément symptomatique de ce phénomène car l'alternance des types fait que trois octets de bourrage sont systématiquement insérés après chaque `char`, ce qui conduit à une structure occupant globalement beaucoup plus d'octets que nécessaire.

Remarquez que la structure *S3* occupe ici trente-deux octets (voir l'exemple précédent), comme si chaque `char` consommait l'espace d'un `int`, alors que le dernier membre est un `char` (donc suivi d'aucun autre membre de type `int`). Les trois derniers octets de bourrage semblent superflus : vingt-neuf octets en tout n'auraient-ils pas été suffisants ? Le compilateur décide cependant de terminer la structure par du bourrage car de telles structures peuvent être placées dans des tableaux : les membres de chacune d'elles doivent être correctement alignés. Puisque chaque élément du tableau occupe la taille d'une structure, il faut arrondir cette taille de telle façon que les éléments successifs conservent leur alignement.

Nous retiendrons que ***la seule chose que nous connaissions sur la disposition des membres d'une structure est leur ordre, mais nous ne pouvons faire aucune supposition quant à l'écart en octet qui les sépare***. Toutefois, nous savons que ***la taille d'une structure est supérieure ou égale à la somme de la taille de ses membres***.

Il est cependant important de chercher à minimiser cette taille, en particulier si nous utilisons des tableaux contenant de nombreuses structures. En effet, l'accès aux données par le processeur est le facteur le plus limitant en performance pour un programme. Sur notre exemple, la structure *S3* est bien plus volumineuse que la structure *S1* alors qu'elle n'apporte aucune information utile supplémentaire : la disposition des membres de la structure *S1* est donc la meilleure dans le cas présent[110]. Une règle qui est suivie par beaucoup de programmeurs pour optimiser ce placement sans dépendre des spécificités d'une plateforme particulière consiste à regrouper les membres par type selon l'ordre supposé de leurs tailles (voir le chapitre 4). Par exemple : tous les `double`, puis les `int`, puis les `char`.

7.2. Usage effectif des structures

Comme indiqué en 7.1.1, une structure permet de regrouper dans une même entité logique des données dont la signification serait confuse si elles étaient éparses. Le service rendu par une structure dépasse généralement la simple mémorisation des données : une structure est

109 Sur certains processeurs, un mauvais alignement produit une dégradation des performances alors que sur d'autres cela provoque un plantage de l'exécution.

110 Il existe des cas très particuliers, concernant des structures très volumineuses, pour lesquels la disposition des membres peut être optimisée en fonction de la fréquence à laquelle les algorithmes accèdent à certains d'entre-eux.

souvent accompagnée de fonctions qui la manipulent pour réaliser des traitements usuels autour de la notion représentée ; nous parlons de la démarche d'***encapsulation*** (vocabulaire). Dans la pratique, la définition d'une structure et des fonctions associées fait l'objet d'un module au sens de la démarche de programmation modulaire étudiée dans l'exercice pratique du chapitre 2. Voici quelques exemples de mise en œuvre.

7.2.1. Un type simple : un peu plus qu'un type de base

Un prétexte très courant pour l'usage d'une structure consiste à définir un type légèrement plus élaboré que les types de base, pour lequel nous fournissons un jeu d'opérations bien spécifiées dans son contexte d'utilisation[111]. Nous reprenons ici le vecteur tridimensionnel déjà abordé. Les déclarations et définitions de cette structure et de quelques fonctions associées, ainsi qu'un exemple simple d'utilisation sont donnés par ces trois fichiers[112].

```
//---- vec3D.h ----
#ifndef VEC3D_H
#define VEC3D_H 1
typedef struct { double x,y,z; } Vec3D;
Vec3D  Vec3D_set  (double x, double y, double z); // (x,y,z) vector
Vec3D  Vec3D_zero (void);                         // (0.0,0.0,0.0) vector
Vec3D  Vec3D_plus (Vec3D v, Vec3D w);             // vector addition: v+w vector
Vec3D  Vec3D_times(Vec3D v, double d);            // vector scaling: v*d vector
double Vec3D_dot  (Vec3D v, Vec3D w);             // dot product: v*w scalar
double Vec3D_mag  (Vec3D v);                      // vector magnitude: scalar
Vec3D  Vec3D_unit (Vec3D v);                      // unit length vector
#endif // VEC3D_H
```

```
//---- vec3D.c ----
#include "vec3D.h"
#include <math.h> // adjust linker settings in GNUmakefile: LDFLAGS+=-lm
Vec3D  Vec3D_set  (double x, double y, double z) { Vec3D v={ x, y, z }; return v; }
Vec3D  Vec3D_zero (void)                         { return Vec3D_set(0.0,0.0,0.0); }
Vec3D  Vec3D_plus (Vec3D v, Vec3D w)             { return Vec3D_set(v.x+w.x,v.y+w.y,v.z+w.z); }
Vec3D  Vec3D_times(Vec3D v, double d)            { return Vec3D_set(v.x*d,v.y*d,v.z*d); }
double Vec3D_dot  (Vec3D v, Vec3D w)             { return v.x*w.x+v.y*w.y+v.z*w.z; }
double Vec3D_mag  (Vec3D v)                      { return sqrt(Vec3D_dot(v,v)); }
Vec3D  Vec3D_unit (Vec3D v)                      { return Vec3D_times(v,1.0/Vec3D_mag(v)); }
```

```
//---- prog.c ----
#include <stdio.h>
#include "vec3D.h"
int main(void)
{
Vec3D v[5]={ Vec3D_zero() };
for(int i=1;i<5;++i)
  { v[i]=Vec3D_plus(v[i-1],Vec3D_times(Vec3D_set(i,i+1,i+2),0.2)); }
Vec3D dir=Vec3D_unit(v[4]);
printf("%g %g %g\n",dir.x,dir.y,dir.z); // 0.40161 0.562254 0.722897
return 0;
}
```

L'inclusion du fichier d'en-tête standard *`math.h`* et la modification du fichier *`GNUmakefile`* sont nécessaires dans cet exemple car nous utilisons la fonction mathématique standard *`sqrt()`* qui calcule la racine carrée d'un réel (voir le chapitre 10). Le type réalisé ici peut être vu comme un type numérique étendu (à trois composantes). Les fonctions associées nous évitent d'avoir à répéter les détails de ces calculs usuels à chaque fois qu'ils sont nécessaires dans l'application.

111 Le langage C++ fournit des moyens pour réaliser très rigoureusement cette démarche.

112 Le style d'écriture est ici très compact afin de conserver une bonne vue d'ensemble ; les détails sont triviaux.

Puisque ce type a une taille raisonnable, nous n'hésitons pas à en recopier les valeurs et à en produire de nouvelles (comme pour les types de base). Avec les compilateurs modernes, qui optimisent le code de manière très agressive, cette solution est en effet plus efficace que la manipulation de structures simples par pointeurs[113] ; ce point sera à nouveau discuté en 7.2.3.

7.2.2. Un type élaboré : gestion de ressources

Un autre prétexte très courant pour l'usage d'une structure consiste à lui confier la gestion d'une ressource. Nous utilisons ici, comme exemple simpliste, un type dédié à la mémorisation d'un nombre variable d'entiers. Les déclarations et définitions de cette structure et de quelques fonctions associées, ainsi qu'un exemple simple d'utilisation sont donnés par ces trois fichiers[114].

```
//---- mem.h ----
#ifndef MEM_H
#define MEM_H 1
typedef struct { int c; int *d; } Mem;
void Mem_init   (Mem *m);                       // set initial state
void Mem_destroy(Mem *m);                       // free resources
int  Mem_count  (const Mem *m);                 // number of values
int  Mem_get    (const Mem *m, int index);      // get value at index
void Mem_set    (Mem *m, int index, int value); // set value at index
void Mem_append (Mem *m, int value);            // memorise a new value
void Mem_show   (const Mem *m);                 // display values
#endif // MEM_H
```

```
//---- mem.c ----
#include "mem.h"
#include <stdlib.h>
#include <stdio.h>
void Mem_init   (Mem *m)                       { m->c=0; m->d=NULL; }
void Mem_destroy(Mem *m)                       { free(m->d); Mem_init(m); }
int  Mem_count  (const Mem *m)                 { return m->c; }
int  Mem_get    (const Mem *m, int index)      { return m->d[index]; }
void Mem_set    (Mem *m, int index, int value) { m->d[index]=value; }
void Mem_append (Mem *m, int value)
                { m->d=(int *)realloc(m->d,(size_t)(1+m->c)*sizeof(int)); m->d[m->c++]=value; }
void Mem_show   (const Mem *m)
                { for(int i=0;i<m->c;++i) { printf("%d ",m->d[i]); } printf("\n"); }
```

```
//---- prog.c ----
#include "mem.h"
int main(void)
{
Mem m; Mem_init(&m);                                    // initialisation
Mem_append(&m,1); Mem_append(&m,2); Mem_append(&m,3); // mémorisation
for(int i=0,count=Mem_count(&m);i<count;++i)          // modification
  { Mem_set(&m,i,Mem_get(&m,i)*10); }
Mem_show(&m);                                           // 10 20 30
Mem_destroy(&m);                                        // libération des ressources
return 0;
}
```

Les fonctions réalisées assurent un invariant (ici le lien entre l'allocation et le nombre d'éléments mémorisés) qui garantit que si la structure est uniquement manipulée par ces fonctions elle restera dans un état cohérent. L'initialisation préalable et la libération finale sont

113 Une habitude historique très répandue consistait à s'interdire la recopie des structures. Il était prétendu que le recours à des indirections via un pointeur était plus efficace ; si cela a pu être vrai un jour, c'est désormais faux !

114 Le style d'écriture est encore ici très compact afin de conserver une bonne vue d'ensemble.

indispensables au bon fonctionnement. Il est très important, lors de l'usage d'un tel type de donnée élaboré, de respecter scrupuleusement l'interface de programmation qui lui est associée. Tout autre intervention non prévue, faisant un usage direct du contenu de la structure, risque de compromettre son intégrité en commettant une infraction à l'invariant.

Voici un programme incorrect qui ne respecte pas l'interface de programmation de ce type.

```
//---- prog.c (incorrect) ----
#include <stdio.h>
#include "mem.h"
int main(void)
{
Mem m1; Mem_init(&m1);                                    // initialisation de m1
Mem_append(&m1,1); Mem_append(&m1,2); Mem_append(&m1,3); // mémorisation dans m1
Mem m2=m1;                                                // copie incorrecte de m1 vers m2 !!!
for(int i=0,count=Mem_count(&m2);i<count;++i)             // modification de m2 (et de m1 !!!)
  { Mem_set(&m2,i,Mem_get(&m2,i)*100); }
printf("m1: "); Mem_show(&m1);                            // m1: 100 200 300
Mem_destroy(&m1);                                         // libération de m1 (et de m2 !!!)
Mem_append(&m2,400); Mem_append(&m2,500);                 // ... la suite travaille sur une zone
printf("m2: "); Mem_show(&m2);                            // mémoire libérée, le comportement est
Mem_destroy(&m2);                                         // indéfini (plantage probable) ...
return 0 ;
}
```

En effet, nous recopions trivialement *m1* vers *m2* ce qui a pour effet de recopier à l'identique chacun des deux membres[115]. L'un d'eux est un pointeur vers un bloc de données alloué dynamiquement ; nous nous retrouvons donc avec les deux structures qui désignent ce même bloc. Quand l'un parcourt son contenu pour le modifier, l'autre qui accède aux mêmes données voit alors son contenu également modifié, ce qui n'est absolument pas l'effet voulu. Lors de la libération des ressources de *m1*, le bloc mémoire qu'il désigne est rendu au système d'exploitation. Puisque *m2* le désigne également, toutes ses futures opérations manipulent des données qui ne devraient plus lui être accessibles. À partir de cet instant le comportement du programme devient indéterminé, c'est à dire que la moindre opération peut provoquer un effet totalement imprévisible (des calculs incorrects, un plantage...).

C'est pour cette raison que, ***lorsque nous réalisons une structure qui gère des ressources (mémoire, fichiers…) nous nous interdisons de la transmettre par recopie : nous nous imposons l'usage systématique de pointeurs***. La maîtrise précise des conséquences d'une copie est trop délicate pour prendre le moindre risque (ce problème ne se pose pas avec de simples données numériques). La plupart des fonctions qui manipulent la structure *Mem* de notre exemple ont une bonne raison de le faire par l'intermédiaire d'un pointeur puisqu'elles visent à modifier une variable située dans le contexte appelant (voir en 3.1.2). En revanche, pour quelques autres (*Mem_count()*, *Mem_get()* et *Mem_show()*) aucune modification de la structure n'est envisagée ; nous nous interdisons malgré tout d'avoir recours à une recopie. Au lieu de cela, ***nous ajoutons le qualificatif*** *const* ***au pointeur passé en paramètre pour signifier au compilateur (et au lecteur du prototype) qu'aucune modification n'est autorisée sur la structure pointée***.

7.2.3. Optimisation de l'interface

S'imposer l'utilisation systématique des fonctions constituant l'interface de programmation associée à une structure présente un intérêt certain en terme de maintenance du code. Même si les exemples choisis ici sont volontairement très simples, il n'est pas du tout rare que beaucoup de fonctions servant à manipuler des structures ne consistent qu'en très peu d'opérations. Les invocations de fonctions ont un coût en temps d'exécution et, dans ce cas, ce dernier peut se révéler très pénalisant devant le temps nécessaire aux opérations réellement utiles.

115 Le langage C++ permet un contrôle très fin, voire une interdiction, de ces opérations délicates.

Pour faire l'économie de ces invocations de fonctions très simples nous pourrions être tentés de recopier leur code à chaque endroit où nous en avons besoin dans l'application. Ce serait une très mauvaise solution, contraire au principe d'encapsulation, puisque la moindre modification dans la logique de fonctionnement de la structure nécessiterait de multiples interventions dans le code source de l'application pour l'adapter en conséquence. Toutefois, elle apporterait deux avantages en terme de performances.

Comme vous le verrez dans les matières S5-MIP et S6-MIP de l'ENIB, l'appel à une fonction consiste en la recopie des paramètres effectifs vers les paramètres formels, en un branchement vers la fonction, et en la sauvegarde de quelques registres du processeur. De manière complémentaire, la sortie d'une fonction consiste en la recopie de son résultat vers le contexte appelant, la restauration des registres sauvegardés et un branchement pour poursuivre l'exécution dans le contexte appelant. Si nous nous débarrassions de ces deux longues phases pour ne conserver que les quelques opérations utiles qu'elles encadrent nous diminuerions sensiblement le nombre d'instructions exécutées par le processeur, ce qui se traduirait par un gain en temps d'exécution.

De manière encore plus significative, il faut savoir que le compilateur optimise le code exécutable en éliminant et en remaniant les opérations et les précautions dont il est capable de prouver la redondance ou l'inutilité. Plus nous lui exposons d'informations sur le contexte d'un traitement plus il est capable de prendre des décisions efficaces. La recopie des seules instructions utiles à notre traitement sur la structure parmi toutes celles du contexte appelant exposerait au compilateur une vision plus large de l'ensemble des traitements sur ces données. Dans ces conditions il aurait bien plus d'éléments de décision que ce qui lui est perceptible depuis l'intérieur de la fonction, ce qui conduirait à une meilleure optimisation du code exécutable produit.

Depuis sa version C99, le langage C propose le mot-clef `inline` qui sert à qualifier une fonction afin que le compilateur remplace les appels à celle-ci par une insertion de son code à l'endroit de l'invocation[116]. Dans ces conditions, les deux avantages précédemment exposés deviennent effectifs tout en conservant le principe d'encapsulation qui assure une bonne maintenabilité du code. Pour que cela soit possible, il ne suffit plus au compilateur de connaître la déclaration de la fonction à l'endroit de l'appel, mais il doit également connaître sa définition complète afin d'en réutiliser le code. Il s'agit d'un changement de stratégie radical vis-à-vis de la démarche de programmation modulaire dans laquelle seules les déclarations des fonctions sont exposées dans les fichiers d'en-tête (voir le chapitre 2).

Désormais, dans un module de code, nous déplacerons vers le fichier d'en-tête la définition des fonctions qui sont qualifiées avec le mot-clef `inline`. Seulement, puisqu'un tel fichier d'en-tête peut être inclus depuis plusieurs unités de compilation, une même fonction `inline` risque de se trouver définie plusieurs fois, dans des modules distincts, au moment de l'édition de liens. Nous lui ajoutons alors le mot-clef `static` (dont l'usage sur une fonction a déjà été rencontré en 6.2) pour signifier que cette fonction ne doit pas être visible à l'extérieur de l'unité de compilation ; ses multiples définitions deviennent ainsi indépendantes du point de vue de l'édition de liens.

Le choix entre les fonctions qui méritent d'être qualifiées avec `inline static` et celles qui garderont leur définition habituelle est une affaire de compromis. Il ne faut pas abuser de cette fonctionnalité car elle risque d'accroître considérablement la taille du code exécutable généré, ce qui peut aboutir à un ralentissement de l'exécution[117]. Ce choix est très pertinent pour les fonctions très courtes qui ne sont constituées que de quelques instructions très simples. Celles qui au contraire sont longues ou qui contiennent d'autres appels coûteux ne tireraient aucun bénéfice de cette possibilité qui alourdirait alors inutilement le code exécutable généré.

116 Cette notion est originellement apparue en C++ et est toujours très activement employée.

117 La raison de ce ralentissement est due à l'épuisement de la mémoire cache dédiée aux instructions ; ce sujet dépasse largement le cadre de cette initiation au langage C.

Voici une nouvelle formulation du fichier d'en-tête dédié aux vecteurs tridimensionnels qui a été présenté en 7.2.1.

```
//---- vec3D.h ----
#ifndef VEC3D_H
#define VEC3D_H 1
#include <math.h> // adjust linker settings in GNUmakefile: LDFLAGS+=-lm
typedef struct { double x,y,z; } Vec3D;
inline static Vec3D  Vec3D_set  (double x, double y, double z) // (x,y,z) vector
                     { Vec3D v={ x, y, z }; return v; }
inline static Vec3D  Vec3D_zero (void)                         // (0.0,0.0,0.0) vector
                     { return Vec3D_set(0.0,0.0,0.0); }
inline static Vec3D  Vec3D_plus (Vec3D v, Vec3D w)             // vector addition: v+w vector
                     { return Vec3D_set(v.x+w.x,v.y+w.y,v.z+w.z); }
inline static Vec3D  Vec3D_times(Vec3D v, double d)            // vector scaling: v*d vector
                     { return Vec3D_set(v.x*d,v.y*d,v.z*d); }
inline static double Vec3D_dot  (Vec3D v, Vec3D w)             // dot product: v*w scalar
                     { return v.x*w.x+v.y*w.y+v.z*w.z; }
inline static double Vec3D_mag  (Vec3D v)                      // vector magnitude: scalar
                     { return sqrt(Vec3D_dot(v,v)); }
inline static Vec3D  Vec3D_unit (Vec3D v)                      // unit length vector
                     { return Vec3D_times(v,1.0/Vec3D_mag(v)); }
#endif // VEC3D_H
```

Dans ce cas simpliste, toutes les fonctions sont qualifiées de *`inline static`* puisqu'elles ne contiennent que quelques opérations élémentaires[118]. Cela confirme le fait que le type inventé ici n'est guère plus qu'un type arithmétique. L'usage des fonctions évite simplement une recopie fastidieuse et source d'erreurs de ce jeu d'opérations usuelles. Le module en question ne contient donc aucun autre code source que ce fichier d'en-tête.

De la même façon, voici une nouvelle formulation du fichier d'en-tête dédié à la mémorisation d'entiers qui a été présenté en 7.2.2.

```
//---- mem.h ----
#ifndef MEM_H
#define MEM_H 1
#inlude <stdlib.h>
typedef struct { int c; int *d; } Mem;
inline static void Mem_init   (Mem *m)                      // set initial state
                   { m->c=0; m->d=NULL; }
              void Mem_destroy(Mem *m);                     // free resources
inline static int  Mem_count  (const Mem *m)                // number of values
                   { return m->c; }
inline static int  Mem_get    (const Mem *m, int index)     // get value at index
                   { return m->d[index]; }
inline static void Mem_set    (Mem *m, int index, int value) // set value at index
                   { m->d[index]=value; }
              void Mem_append (Mem *m, int value);          // memorise a new value
              void Mem_show   (const Mem *m);               // display values
#endif // MEM_H
```

Ici, seules les fonctions dont le code ne contient que peu d'instructions simples ont été qualifiées de *`inline static`*. Bien que nous ne fassions dans l'immédiat aucun appel aux fonctions déclarées dans le fichier d'en-tête standard *`stdlib.h`* (nous le ferons dans les autres fonctions), nous l'incluons pour rendre la *macro* *`NULL`* accessible.

118 Même la racine carrée est une simple instruction ; voir l'outil https://godbolt.org/ .

Le fichier de définition, quant à lui, se retrouve soulagé des fonctions que nous avons précédemment déplacées.

```
//---- mem.c ----
#include "mem.h"
#include <stdio.h>
void Mem_destroy(Mem *m)
{ free(m->d); Mem_init(m); }
void Mem_append (Mem *m, int value)
{ m->d=(int *)realloc(m->d,(size_t)(1+m->c)*sizeof(int)); m->d[m->c++]=value; }
void Mem_show(const Mem *m)
{ for(int i=0;i<m->c;++i) { printf("%d ",m->d[i]); } printf("\n"); }
```

Il ne reste ici que les fonctions qui font usage des fonctionnalités standards d'allocation dynamique de mémoire et d'affichage ; elles sont effectivement réputées coûteuses en temps d'exécution.

L'utilisation qui peut être faite de la nouvelle formulation de ces trois fichiers est strictement identique à celle que nous faisions de leur version précédente.

En 7.2.1 et 7.2.2 nous discutions également des critères pour décider de transmettre les structures par valeur ou par adresse aux fonctions. Au delà de ce que ces choix impliquent en terme d'expressivité (possibilité de modifier ou non une variable du contexte appelant), nous nous interrogions sur le coût des recopies par rapport à celui des indirections. Dans le cas des fonctions `inline` ce problème de coût est complètement évacué puisqu'il n'y aura aucun appel donc aucun passage de paramètre. Le compilateur aura directement connaissance du paramètre effectif : il provoquera sa modification directe si c'était l'effet attendu d'un passage par adresse et s'en gardera dans le cas d'un passage par valeur.

7.3. Résumé

Le propos de ce cours était d'introduire la notion de structure de données. Il s'agit d'un moyen de définir de nouveaux types de données par composition de types existants. Au delà des aspects syntaxiques qui permettent de définir les membres d'une telle structure, le moyen de les initialiser et d'y accéder, nous avons vu que des données de ce type pouvaient être recopiées aussi trivialement qu'avec les types de base. Une attention particulière a été portée sur l'influence de la disposition des membres d'une structure envers la place qu'elle occupe en mémoire.

Dans un second temps, nous nous sommes concentrés sur les bonnes pratiques qui concernent l'usage de telles structures. Nous avons vu en particulier qu'elles étaient étroitement liées à la notion de programmation modulaire. Un tel module est consacré à la mise à disposition de fonctions formant une interface de programmation décrivant le jeu d'opérations usuelles sur le nouveau type de données introduit ; il s'agit de l'encapsulation. Deux cas extrêmes ont été choisis pour représenter d'une part un type très simple apportant une facilité d'écriture mais ne nécessitant pas de précaution particulières, et d'autre part un type plus élaboré devant garantir la bonne gestion de ressources internes et pour lequel, au contraire, il était indispensable de s'astreindre à un usage strict de l'interface de programmation fournie afin d'éviter des incohérences.

Le recours systématique à une telle interface de programmation assure une bonne maintenabilité du code mais peut introduire un coût en temps d'exécution. L'usage du qualificatif `inline` est alors présenté afin d'éliminer, dans les cas qui le méritent, un tel surcoût tout en conservant la maintenabilité offerte par l'encapsulation.

N.B. : Il existe dans le langage C d'autres constructions syntaxiques qui ressemblent à celle des structures. Leur usage est bien moins courant que celui des structures ; nous ne les présenterons pas dans le cadre de ce cours. Voici en résumé ce dont il s'agit :

- Une `union` représente en quelque sorte une structure dont tous les champs sont superposés à la même adresse en mémoire.
  (http://en.cppreference.com/w/c/language/union)

- Un champ de *bits* (*bit-field*) est une structure dans laquelle nous attribuons un nom à quelques *bits* seulement d'un membre entier.
  (http://en.cppreference.com/w/c/language/bit_field)
- Une *enum* sert à inventer un type dont l'étendue des valeurs est un ensemble de constantes entières.
  (http://en.cppreference.com/w/c/language/enum)

8. L04_IO : Entrées-sorties

Motivation :

Tous les exemples et exercices traités jusqu'alors rendent leurs résultats visibles grâce à la fonction `printf()` qui produit du texte dans le terminal. Il ne s'agit que d'un cas particulier des fonctionnalités d'entrées-sorties proposées par la bibliothèque standard du langage C.

Ces fonctionnalités concernent d'une part le formatage de données sous forme de texte, que ce soit pour le produire ou l'analyser, et d'autre part l'envoi et la réception d'un tel texte ou de données binaires sur des flux sortants ou entrants.

Consignes :

Au delà de la découverte très guidée de ces quelques nouvelles fonctionnalités, le travail demandé ici nécessitera d'être autonome dans la réutilisation des notions déjà acquises (allocation, passages de paramètres...) afin de les appliquer à un cas d'usage concret. Il faudra également savoir consulter la documentation. Veilliez à suivre scrupuleusement chacune des étapes en vous assurant à chaque fois d'avoir bien observé et compris ce qui vous est décrit.

8.1. Mise en place du programme

Ouvrez un terminal et grâce à la ligne de commande du *shell*, placez vous dans le répertoire de travail choisi pour ce sujet. Saisissez par exemple :

```
$ mkdir S3PRC_L04_IO ↲
$ cd S3PRC_L04_IO ↲
```

Placez dans ce répertoire le fichier `GNUmakefile` générique mis à votre disposition.

(http://www.enib.fr/~harrouet/Data/Courses/GNUmakefile)

Nous suivrons en effet la démarche de programmation modulaire présentée au chapitre 2.

Le programme principal sera rédigé dans le fichier `prog04.c` et appellera une fonction `test_output()` qui sera réalisée dans ce même fichier. Cette dernière qui, n'attend aucun paramètre et ne renvoie aucun résultat, se contentera pour l'instant d'afficher son nom puis, dans un unique message mais de manière clairement lisible, les valeurs remarquables que vous aurez choisi d'attribuer à quatre variables de type entier, réel, caractère et chaîne.

Fabriquez ce programme et exécutez-le afin de vérifier que tout est en place. En l'état, il ne fait qu'afficher le nom de la fonction `test_output()` et les valeurs que vous avez choisies :

```
$ make ↲
$ ./prog04 ↲
```

8.2. Formatage de texte

Les fonctionnalités de formatage que nous utilisons dans les invocations de la fonction `printf()` restent accessibles même si nous ne souhaitons pas produire immédiatement un message dans le terminal. À cet effet, la fonction `sprintf()` s'utilise de la même façon, si ce

n'est qu'elle produit son résultat dans une chaîne de caractères que nous pouvons utiliser comme bon nous semble *a posteriori*.

(http://en.cppreference.com/w/c/io/fprintf)[119]

Intervenez alors dans la fonction `test_output()` pour remplacer l'écriture des valeurs des variables dans le terminal par leur écriture vers une chaîne à l'aide de la fonction `sprintf()`. Il suffit d'indiquer cette chaîne en paramètre avant la chaîne de format. Bien entendu, la chaîne devant accueillir le message produit devra être suffisamment largement dimensionnée pour éviter tout risque de débordement de capacité. Vous devrez ensuite afficher dans le terminal la chaîne produite comme nous en avons désormais l'habitude.

Fabriquez à nouveau le programme et exécutez-le afin de constater que la chaîne composée a bien la forme attendue.

Dans la pratique, il est assez souvent probable que ***la production d'une chaîne formatée risque d'atteindre une longueur excessive par rapport à l'espace de stockage réservé ; il est alors recommendé de recourir à la fonction*** `snprintf()`. Elle attend en effet comme deuxième paramètre la capacité de la chaîne à rédiger, ce qui permet de s'interdire de la dépasser ; la chaîne est simplement tronquée et contient malgré tout le zéro terminal. Modifiez la fonction `test_output()` pour faire usage de cette variante, fabriquez à nouveau le programme et exécutez-le afin de vérifier que rien n'a changé. Reprenez alors cette démarche avec une capacité légèrement trop petite qui permettra de constater la troncature.

8.3. Sortie standard et sortie d'erreurs standard

Les opérations d'entrées sorties s'appuient sur des ***flux*** (vocabulaire), c'est à dire des objets qui sont reliés à un moyen de communication (fichier, terminal...), qui permettent d'en connaître l'état et qui proposent des fonctionnalités de formatage ou encore de mise en tampon (*buffer*) afin de limiter la sollicitation trop fréquente des ressources matérielles. Ils sont représentés dans les programmes par des pointeurs sur une donnée de type `FILE`. Il s'agit d'un ***type opaque*** (vocabulaire), c'est à dire que nous le désignons par un pointeur sans même savoir comment il est constitué ; seules les fonctions de la bibliothèque standard en connaissent les détails.

De tels flux peuvent être ouverts à la demande mais il en existe qui le sont déjà dès le démarrage du programme, ce sont les entrées-sorties standards. Le fichier d'en-tête standard `stdio.h` déclare à ce propos deux variables globales : `stdout` pour la sortie standard et `stderr` pour la sortie d'erreurs standard. Puisqu'elles sont globales, ces deux variables sont accessibles depuis toutes les fonctions du programme. Elle désignent respectivement le flux pour produire les messages qui sont normalement destinés à être lus par l'utilisateur du programme, et le flux pour signaler les anomalies.

Complétez la fonction `test_output()` pour ajouter un affichage de la chaîne rédigée selon le format `"out: <%s>\n"` sur `stdout` et `"err: <%s>\n"` sur `stderr`. Il suffit pour cela d'utiliser la fonction `fprintf()` en lui indiquant le flux concerné comme premier paramètre.

Fabriquez à nouveau le programme et exécutez-le afin d'observer les nouveaux messages produits. Que ce soient les précédents ou ceux-ci, tous ces messages aboutissent dans le terminal. Relancez alors le programme selon les lignes de commande suivantes :

```
$ ./prog04 >output_stdout.txt ↲
$ ./prog04 2>output_stderr.txt ↲
```

Dans le premier cas vous devez constater que seul le message produit sur `stderr` apparaît dans le terminal ; tous les autres ont abouti dans le fichier `output_stdout.txt` que vous pouvez visualiser avec votre éditeur de texte. Dans le second cas, c'est tout l'opposé : le terminal affiche tous les messages sauf celui produit sur `stderr` qui se retrouve dans le fichier `output_stderr.txt`.

---

119 Une unique page de documentation concerne des fonctions semblables (`printf()`, `sprintf()`, `fprintf()`...).

Les deux opérations effectuées ici sont des redirections d'entrées-sorties. Cela n'a rien à voir avec le langage C et concerne essentiellement le système d'exploitation. Comme vérifié ici, la redirection `>` sert à signifier que la sortie standard désignée par `stdout` n'est plus reliée au terminal mais à un fichier pour y écrire. La redirection `2>` fait de même pour la sortie d'erreurs standard désignée par `stderr`.

Cette expérience montre également qu'il n'y a aucune différence entre l'usage de `printf()` et celui de `fprintf()` vers le flux `stdout`. La fonction `printf()` n'est en effet qu'une facilité d'écriture pour ce cas d'usage extrêmement fréquent.

8.4. Écriture de texte dans un fichier

Nous venons de voir que l'écriture de texte dans un flux est tout à fait similaire à ce que nous faisions déjà avec `printf()`. Nous n'avons pas eu à nous soucier de l'ouverture et de la fermeture des flux `stdout` et `stderr` puisqu'ils sont censés exister pendant toute la durée d'exécution du programme. En revanche, si un flux doit désigner un fichier choisi par notre programme, il est nécessaire de l'ouvrir explicitement.

La fonction standard `fopen()` répond à ce besoin.

(http://en.cppreference.com/w/c/io/fopen)

Son premier paramètre désigne le chemin qui mène au fichier que nous voulons désigner dans l'arborescence du système de fichiers. Son deuxième paramètre sert à préciser le mode d'ouverture pour ce fichier. Si l'ouverture se déroule comme prévu, cette fonction renvoie un pointeur sur un flux qu'il nous faudra mémoriser ; en effet c'est un tel pointeur qui est attendu comme premier paramètre de la fonction `fprintf()`. En revanche, en cas d'impossibilité, un pointeur nul est renvoyé. Le mode d'ouverture est une chaîne qui indique ce que nous souhaitons faire avec le flux ouvert ; dans le cas présent nous ne voulons qu'y écrire, ce qui correspond au mode `"w"`.

Chaque flux ouvert explicitement doit être fermé lorsque le programme n'a plus besoin d'y faire référence. Sans cette précaution, un programme qui ouvrirait à de très nombreuses reprises des flux sans les fermer finirait par ne plus pouvoir en ouvrir (ce sont des ressources limitées[120]). La fonction standard `fclose()` sert à réaliser une telle fermeture.

(http://en.cppreference.com/w/c/io/fclose)

Il faut bien entendu lui transmettre le pointeur obtenu avec `fopen()` et il va sans dire que ce pointeur ne doit plus être utilisé après.

Pour tester l'écriture de texte dans un fichier, ajoutez dans le fichier `prog04.c` une fonction `test_txtWrite()` qui sera appelée par le programme principal. Elle ne renvoie aucun résultat mais attend comme paramètre une chaîne de caractères spécifiant le chemin vers le fichier dans lequel nous souhaitons écrire : nous choisirons ici `"output_text.txt"`. Faites en sorte qu'elle commence par afficher son nom et son paramètre. Elle doit ensuite ouvrir en écriture (mode d'ouverture `"w"`) le fichier dont le nom est transmis en paramètre. Si cette ouverture échoue, il faut signaler cet événement par un message explicite dans `stderr` et terminer la fonction. Ce flux servira à écrire des messages contenant des informations de types variés (comme dans `test_output()`) puis devra être fermé.

Fabriquez à nouveau le programme et exécutez-le puis ouvrez, dans votre éditeur de texte, le fichier produit afin de vérifier que le contenu attendu y est bien inscrit.

Complétez encore le programme principal par un second appel à cette même fonction mais en demandant cette fois l'écriture dans le fichier `"/output_forbidden.txt"`. Ce chemin indique que le fichier doit se trouver à la racine du système de fichiers. À moins que vous soyez l'administrateur du poste informatique, vous n'avez normalement pas le droit de créer un fichier à cet emplacement et vous devez constater que, lors de cette nouvelle tentative, l'ouverture du flux échoue.

120 La limitation ne concerne pas directement les flux mais les descripteurs de fichier. Ce sont des ressources dédiées à la communication, gérées par le système d'exploitation, sur lesquelles les flux s'appuient pour fonctionner.

8.5. Sauvegarde d'une image dans un format textuel

Nous réutilisons les connaissances précédentes pour les appliquer à un cas concret. Il s'agit de sauvegarder une image dans un fichier selon un format très simple constitué exclusivement de texte : la variante P3 du format Netpbm.

(https://en.wikipedia.org/wiki/Netpbm_format)

Pour ceci, il faut créer un nouveau module constitué du fichier *netpbm.c* et de son fichier d'en-tête *netpbm.h* en respectant les consignes usuelles. Il faudra déclarer et définir la fonction *Netpbm_saveP3()* qui attend en paramètres le nom du fichier dans lequel l'image sera sauvegardée, deux entiers pour la largeur et la hauteur de l'image, et un pointeur sur des *uint8_t* qui constituent le contenu de l'image ; elle doit renvoyer un booléen indiquant le succès de la sauvegarde.

La définition de cette fonction doit commencer par ouvrir un flux en écriture sur le fichier spécifié. Si cette opération échoue, il faut terminer la fonction en renvoyant *false* pour signaler l'échec. Comme indiqué dans la documentation, le préambule de la variante P3 du format Netpbm commence par le préfixe *P3* sur la première ligne de texte, puis la largeur et la hauteur sur la deuxième et enfin la valeur *255* sur la troisième. Le nombre de pixels de l'image correspond au produit de sa largeur par sa hauteur, et chacun d'eux est constitué de trois composantes de couleur (rouge, verte, bleu, valant de *0* à *255*) ce qui donne le nombre des *uint8_t* décrivant l'image. Il suffit d'écrire chacun d'eux à la suite du préambule (éventuellement en changeant de ligne de temps en temps pour un disposition plus harmonieuse, mais ce n'est pas indispensable). Il ne reste plus qu'à fermer le flux sur ce fichier et signifier le succès de la sauvegarde en renvoyant *true*.

Une fonction *test_saveP3()*, attendant en paramètres une chaîne *fileName* et deux entiers *width* et *height*, doit être réalisée dans le fichier *prog04.c* et sera appelée par le programme principal en utilisant "*output_A.ppm*", *320* et *200* comme paramètres effectifs. Après l'affichage de son nom et de ses paramètres, elle doit allouer dynamiquement un tableau *data* de *3*width*height* éléments de type *uint8_t* et les initialiser de cette façon :

```
for(int y=0;y<height;++y)
  {
  for(int x=0;x<width;++x)
    {
    const int red=255*x/(width-1);
    const int green=255*y/(height-1);
    const int blue=255-(red+green)/2;
    const int idx=3*(y*width+x);
    data[idx+0]=(uint8_t)red;
    data[idx+1]=(uint8_t)green;
    data[idx+2]=(uint8_t)blue;
    }
  }
```

Les composantes rouges varient de *0* à *255* selon la largeur de l'image et les composantes vertes font de même selon la hauteur. Les composantes bleues sont les complémentaires de la moyenne des deux autres. Faites l'effort de comprendre les détails de ces boucles ; en particulier le calcul de *idx* traduit le fait que les pixels sont ordonnés (de gauche à droite) au sein de chaque ligne et que les lignes s'enchaînent (de haut en bas) dans le tableau.

Il ne reste plus qu'à sauvegarder cette image par un appel à la fonction *Netpbm_saveP3()*, en produisant un message d'erreur sur *stderr* en cas d'échec, et à libérer la mémoire allouée dynamiquement.

Fabriquez à nouveau le programme et exécutez-le. La visualisation du fichier *output_A.ppm* (en cliquant dessus par exemple) doit représenter une image correspondant aux dégradés de couleurs attendus. En ouvrant ce même fichier dans un éditeur de texte vous devez également y lire des informations compréhensibles (notamment le préambule).

8.6. Entrée standard

De la même façon qu'il existe des sorties standards déjà ouvertes au lancement du programme, le fichier d'en-tête standard *stdio.h* rend accessible, par la déclaration de la variable globale *stdin*, un troisième flux implicitement ouvert sur l'entrée du terminal. Il permet à l'utilisateur du programme de saisir des informations au clavier.

Le moyen le plus commun pour réaliser de telles saisies est la fonction *scanf()* ; dans le même esprit que *printf()*, il ne s'agit que d'une économie d'écriture pour invoquer la fonction *fscanf()* sur le flux *stdin*.

(http://en.cppreference.com/w/c/io/fscanf)

Elle présente des similitudes avec la fonction *printf()* dans la mesure où on lui fournit une chaîne de format dont les symboles % se réfèrent aux paramètres qui la suivent. Seulement ici le rôle de cette fonction est de renseigner les *lvalues* désignées par ces paramètres à partir de ce qui est analysé en entrée du flux ; ce sont donc des adresses.

Une fonction *test_input()*, n'attendant aucun paramètre et ne renvoyant aucun résultat doit être réalisée dans le fichier *prog04.c* et sera appelée par le programme principal. Elle commencera par afficher son nom puis utilisera les instructions suivantes.

```
char fileName[100]="";
int width=0, height=0;
printf("file name?\n");
scanf("%s",fileName);
printf("image size (width x height)?\n");
scanf("%d x%d",&width,&height);
if(fileName[0]&&(width>0)&&(height>0))
  { test_saveP3(fileName,width,height); }
```

Le premier appel à *scanf()* utilise le nom du tableau *fileName* qui représente bien l'adresse de son premier élément. La chaîne de format *"%s"* indique qu'il faut reconnaître un mot, c'est à dire une séquence de caractères qui ne sont pas des séparateurs, au sens de *isspace()* (voir en 5.1.3), qu'il faudra stocker à l'adresse indiquée (avec un zéro terminal). La chaîne de format *"%d x%d"* du second appel à *scanf()* réclame l'extraction de deux entiers séparés par le texte littéral *"x"* ; ils seront stockés dans les variables *width* et *height* dont les adresses sont indiquées à la suite. Les espaces qui figurent entre les éléments d'une chaîne de format correspondent à une séquence quelconque, éventuellement vide, de séparateurs au sens de *isspace()* (voir en 5.1.3). La plupart des symboles % consomment les éventuels séparateurs qui les précèdent. Ceci fait qu'en pratique, sur cet exemple, des séparateurs peuvent précéder et suivre, ou non, les extractions et le texte *"x"* littéral.

À l'issue de cette séquence, les variables *fileName*, *width* et *height* sont renseignées au clavier par l'utilisateur[121] et servent à invoquer à nouveau la fonction *test_saveP3()*.

Fabriquez à nouveau le programme et exécutez-le. À l'invitation, saisissez *output_B.ppm* comme nom de fichier et *320x200* par exemple comme dimensions. Le nouveau fichier qui est obtenu doit être similaire au précédent.

Remarquez qu'aucune erreur de saisie n'est permise. Relancez l'expérience et au moment de la saisie des dimensions fournissez un texte qui ne peut pas être analysé comme un entier (une lettre par exemple). Vous devriez constater que les variables ne sont pas altérées et donc que l'invocation de *test_saveP3()* n'a pas lieu.

Il pourrait être tentant de faire une boucle sur chaque invocation de *scanf()* jusqu'à constater que les valeurs attendues sont effectivement extraites, seulement ce n'est pas suffisant. En effet, lorsque *scanf()* constate que l'extraction n'est pas possible pour le format spécifié avec le symbole %, les caractères inattendus ne sont pas consommés et restent les prochains disponibles dans le flux. Si nous tentions immédiatement une nouvelle invocation, alors nous rencontrerions à nouveau les mêmes caractères inattendus.

121 Faites l'effort de comprendre ce qui est testé dans la condition de l'alternative.

Lorsque l'extraction a lieu de manière interactive, il est probable que des erreurs de saisie se produisent. Une démarche prudente consiste à extraire systématiquement ce qui est saisi pour éventuellement proposer une nouvelle tentative en cas d'erreur. Il suffit pour ceci de commencer par extraire une ligne de texte, puis, après seulement, en analyser le contenu. Si ce dernier ne convient pas il suffit de consommer une nouvelle ligne de texte. La saisie d'une telle ligne s'obtient avec la fonction standard `fgets()`.

(http://en.cppreference.com/w/c/io/fgets)

Elle se charge de remplir une chaîne de caractères (avec un zéro terminal) en prenant soin de ne pas dépasser sa capacité annoncée. Néanmoins, si un retour à la ligne est saisi avant que la capacité de la chaîne soit atteinte (ce qui est le cas généralement attendu), la saisie de la ligne courante s'arrête à ce point. Lorsqu'une ligne est extraite du flux, la fonction `fgets()` renvoie l'adresse de la ligne renseignée. En revanche, lorsqu'il n'est plus possible d'extraire des informations du flux, un pointeur nul est renvoyé.

L'analyse de la ligne extraite repose sur la fonction `sscanf()` qui reprend les mêmes principes que `scanf()` ou `fscanf()` mais analyse la chaîne qui lui est transmise en argument. Un moyen de savoir si les extractions attendues ont eu lieu consiste à contrôler la valeur de retour d'une telle fonction. Elle indique le nombre de symboles `%` qui ont donné lieu à l'extraction attendue. Dans le cas de notre problème, ce résultat doit être `1` pour le nom de fichier et `2` pour les dimensions.

Vous pouvez désormais reformuler la fonction `test_input()` avec une première boucle `do-while` qui demande la saisie d'une ligne puis son analyse jusqu'à ce que le nom de fichier soit extrait, et une seconde boucle qui fait de même pour les dimensions de l'image. Bien entendu, si la saisie d'une ligne devient impossible, il faut terminer cette fonction.

Fabriquez à nouveau le programme et exécutez-le pour vérifier qu'il réitère les invitations en cas de mauvaise saisie. Mis à part cela il doit se comporter comme précédemment.

8.7. Lecture d'une image dans un format textuel

Nous réutilisons nos connaissances précédentes pour les appliquer à un cas concret. Il s'agit de lire une image depuis un fichier selon un format très simple constitué exclusivement de texte : la variante P3 du format Netpbm.

(https://en.wikipedia.org/wiki/Netpbm_format)

Il faut pour ceci compléter le module constitué du fichier `netpbm.c` et de son fichier d'en-tête `netpbm.h` en y déclarant et définissant la fonction `Netpbm_loadP3()` qui attend en paramètres le nom du fichier depuis lequel l'image sera lue et le moyen de mettre à jour deux entiers du contexte appelant qui accueilleront les dimensions de l'image lue ; elle doit renvoyer un pointeur sur les `uint8_t` alloués dynamiquement qui constituent les données de cette image.

La définition de cette fonction doit commencer par ouvrir un flux en lecture sur le fichier spécifié. Il suffit pour cela de réutiliser la fonction standard `fopen()` avec le mode d'ouverture `"r"`. Si cette opération échoue, il faut terminer la fonction en renvoyant un pointeur nul pour signaler l'échec. Comme indiqué dans la documentation, le préambule de la variante P3 du format Netpbm commence par les deux caractères littéraux `'P'` et `'3'`. La fonction standard `fgetc()` fournit un moyen simple d'extraire le prochain caractère d'un flux.

(http://en.cppreference.com/w/c/io/fgetc)

Invoquez donc deux fois cette fonction en vérifiant qu'on obtient bien les caractères attendus. Il faudra ensuite invoquer la fonction `fscanf()` pour extraire trois entiers[122]. Vérifiez que cet appel a bien réalisé trois extractions, que les deux premiers entiers sont strictement positifs et que le troisième vaut `255`. Si une de ces six vérifications échoue, fermez le flux et terminez la fonction par le renvoi d'un pointeur nul.

Les deux premiers entiers extraits représentent respectivement la largeur et la hauteur de l'image. Allouez dynamiquement suffisamment de `uint8_t` pour en mémoriser tous les pixels, sans oublier que chacun d'eux est constitué de trois composantes (rouge, verte et bleue). Chaque `uint8_t` sera renseigné par l'extraction d'un entier depuis le flux. La fonction `fscanf()`

122 En toute rigueur, le préambule peut contenir des commentaires que nous n'envisageons pas ici.

pourra utiliser le format `"%d"` que vous connaissez déjà pour renseigner un `int` qui sera ensuite recopié dans le tableau[123]. Si une de ces multiples extractions échoue, il faudra libérer le tableau obtenu dynamiquement, fermer le flux et terminer la fonction par le renvoi d'un pointeur nul. Il ne reste plus alors qu'à fermer le flux, mettre à jour les entiers du contexte appelant avec la largeur et la hauteur de l'image, avant de renvoyer le tableau décrivant son contenu.

Une fonction `test_loadP3()`, attendant en paramètres deux chaînes `inputFileName` et `outputFileName` doit être réalisée dans le fichier `prog04.c` et sera appelée par le programme principal en utilisant `"output_A.ppm"` et `"output_C.ppm"` comme paramètres effectifs. Après l'affichage de son nom et de ses paramètres, elle doit invoquer la fonction `Netpbm_loadP3()` afin d'obtenir les dimensions et le contenu de l'image indiquée par le paramètre `inputFileName`. En cas d'échec du chargement, il faut produire un message d'erreur sur `stderr` et quitter la fonction. Si l'image est chargée, ses dimensions devront être affichées dans le terminal et tous ses pixels devront être transformés par une permutation des composantes : bleu devient rouge, rouge devient vert, vert devient bleu.

Il ne reste plus qu'à sauvegarder cette image, sous le nom que `outputFileName` indique, par un appel à la fonction `Netpbm_saveP3()`, en produisant un message d'erreur sur `stderr` en cas d'échec, et à libérer la mémoire allouée dynamiquement (obtenue lors du chargement).

Fabriquez à nouveau le programme et exécutez-le. La visualisation du fichier `output_C.ppm` (en cliquant dessus par exemple) doit représenter une image dont les dimensions sont les mêmes que celles de `output_A.ppm` mais dont les dégradés de couleurs ont été permutés.

8.8. Entrées-sorties dans un format binaire

Que ce soit en interaction avec le terminal, ou pour produire et consulter des fichiers, toutes les opérations d'entrées-sorties utilisées jusque là manipulaient du texte. Cela semble naturel pour du texte littéral produit ou lu par un être humain, mais lorsqu'il est question de données numériques, celles-ci doivent être transformées, lors de l'écriture, en une séquence de caractères numériques qui sera analysée, lors de la lecture, pour en retrouver la valeur initiale.

Si ces transformations ont du sens lorsqu'un être humain doit interagir directement avec le texte en question, elles constituent une complication et une charge de travail inutiles lorsque seules les machines informatiques sont concernées. Il existe alors un moyen alternatif pour sauvegarder et relire des données directement dans la forme qu'elles ont dans la mémoire de la machine[124] : le format binaire[125].

8.8.1. Lecture et écriture de données binaires dans un fichier

Pour expérimenter cette nouvelle possibilité, ajoutez au fichier `prog04.c` une fonction `test_binWrite()`, attendant en paramètre une chaîne `fileName` et qui sera appelée par le programme principal en utilisant `"output_data.bin"` comme paramètre effectif. Après l'affichage de son nom et de son paramètre, elle doit créer un tableau de cinq entiers et un tableau de quatre réels dont vous choisirez explicitement toutes les valeurs ; affichez-les alors dans le terminal.

Nous souhaitons sauvegarder ces neuf valeurs dans le fichier dont le nom est donné par le paramètre `fileName`. Vous utiliserez donc à nouveau la fonction standard `fopen()` mais avec le mode d'ouverture `"wb"` ; l'indicateur `"b"` sert à préciser que le contenu sera binaire et non textuel. Cet indicateur n'a généralement aucun effet sur la très grande majorité des systèmes d'exploitation, seul Windows opère une transformation automatique des fins de lignes textuelles : il s'agit désormais de l'en empêcher. En cas d'échec de l'ouverture du flux, il faut produire un message d'erreur sur `stderr` et quitter la fonction.

(http://en.cppreference.com/w/c/io/fopen)

123 Un autre moyen consiste à utiliser le format `"%hhu"` pour spécifier que l'entier extrait est directement un des `uint8_t` (c'est à dire `unsigned char`) du tableau.

124 Dans les matières S5-MIP, S6-MIP et S7-CRS de l'ENIB vous apprendrez qu'il faut toutefois prendre quelques précautions lorsque les données doivent circuler d'une machine à une autre.

125 Ce nom est un abus de langage ici ; la numération n'a pas plus lieu en base deux qu'en aucune autre.

La fonction standard `fwrite()` sert à transférer un bloc d'octets, un tableau par exemple, de la mémoire vers le flux. Le deuxième paramètre indique la taille d'un élément et le troisième leur nombre ; la quantité totale d'octets transmise sera donc le produit de ces deux entiers. Si l'opération se déroule comme attendu, le résultat correspond au nombre d'éléments effectivement transmis.

(http://en.cppreference.com/w/c/io/fwrite)

Utilisez donc cette fonction pour inscrire dans le flux chacun des deux tableaux, en vérifiant à chaque fois que le nombre d'éléments inscrits est correct. Si ce n'est pas le cas, il faut produire un message d'erreur sur `stderr`, fermer le flux et quitter la fonction. Il ne reste plus qu'à fermer le flux.

De façon complémentaire, ajoutez au fichier `prog04.c` une fonction `test_binRead()`, attendant en paramètre une chaîne `fileName` et qui sera appelée par le programme principal en utilisant `"output_data.bin"` comme paramètre effectif. Après l'affichage de son nom et de son paramètre, elle doit créer un tableau de cinq entiers et un tableau de quatre réels dont les contenus resteront indéterminés.

Nous souhaitons initialiser ces neuf valeurs depuis le fichier dont le nom est donné par le paramètre `fileName`. Vous utiliserez donc à nouveau la fonction standard `fopen()` mais avec le mode d'ouverture `"rb"` pour la même raison que dans la fonction précédente. En cas d'échec de l'ouverture du flux, il faut produire un message d'erreur sur `stderr` et quitter la fonction. La fonction standard `fread()` sert à transférer un bloc d'octets, un tableau par exemple, du flux vers la mémoire. La manière de l'utiliser et d'interpréter son résultat est semblable à ce que nous connaissons avec la fonction `fwrite()`. Utilisez donc cette fonction pour remplir chacun des deux tableaux à partir du flux, en vérifiant à chaque fois que le nombre d'éléments extraits est correct. Si ce n'est pas le cas, il faut produire un message d'erreur sur `stderr`, fermer le flux et quitter la fonction. Il ne reste plus qu'à fermer le flux puis afficher le contenu des tableaux dans le terminal.

Fabriquez à nouveau le programme et exécutez-le. Le contenu du fichier `output_data.bin` produit n'est qu'une séquence d'octets incompréhensible pour nous. Toutefois, vous devrez constater que la fonction `test_binRead()` a bien obtenu dans ses tableaux les valeurs qui avaient été choisies dans les tableaux de la fonction `test_binWrite()`.

8.8.2. Sauvegarde d'une image dans un format binaire

Nous réutilisons les connaissances précédentes pour les appliquer à un cas concret. Il s'agit de sauvegarder une image dans un fichier selon un format très simple constitué de texte et de données binaires : la variante P6 du format Netpbm.

(https://en.wikipedia.org/wiki/Netpbm_format)

Il faut pour ceci compléter le module constitué du fichier `netpbm.c` et de son fichier d'en-tête `netpbm.h` en y déclarant et définissant la fonction `Netpbm_saveP6()` ayant le même prototype que la fonction `Netpbm_saveP3()`.

La définition de cette fonction doit commencer par ouvrir un flux en écriture binaire sur le fichier spécifié. Si cette opération échoue, il faut terminer la fonction en renvoyant `false` pour signaler l'échec. Vient ensuite l'écriture du préambule constitué de trois lignes textuelles (c'est autorisé même sur un flux binaire) similaires au cas `P3` si ce n'est que le préfixe de la première ligne sera `P6`. Dans cette variante, l'écriture de l'ensemble des `uint8_t` a lieu de manière binaire : un seul appel à `fwrite()` est suffisant. Il ne reste plus qu'à fermer le flux et signifier le succès de la sauvegarde en renvoyant `true`.

Une fonction `test_saveP6()`, attendant en paramètres deux chaînes `inputFileName` et `outputFileName` doit être réalisée dans le fichier `prog04.c` et sera appelée par le programme principal en utilisant `"output_A.ppm"` et `"output_D.ppm"` comme paramètres effectifs. Après l'affichage de son nom et de ses paramètres, elle doit invoquer la fonction `Netpbm_loadP3()` afin d'obtenir les dimensions et le contenu de l'image indiquée par le paramètre `inputFileName`. En cas d'échec du chargement, il faut produire un message d'erreur sur

stderr et quitter la fonction. Si l'image est chargée, ses dimensions devront être affichées dans le terminal et tous ses pixels devront être transformés par une permutation des composantes : rouge devient bleu, vert devient rouge, bleu devient vert.

Il ne reste plus qu'à sauvegarder cette image, sous le nom que *outputFileName* indique, par un appel à la fonction *Netpbm_saveP6()*, en produisant un message d'erreur sur *stderr* en cas d'échec, et à libérer la mémoire allouée dynamiquement (obtenue lors du chargement).

Fabriquez à nouveau le programme et exécutez-le. La visualisation du fichier *output_D.ppm* (en cliquant dessus par exemple) doit représenter une image dont les dimensions sont les mêmes que celles de *output_A.ppm* mais dont les dégradés de couleurs ont été permutés.

8.8.3. Lecture d'une image dans un format binaire

De la même façon, nous allons procéder à la lecture d'un fichier binaire représentant une image selon la variante P6 du format Netpbm.

Il faut pour ceci compléter le module constitué du fichier *netpbm.c* et de son fichier d'en-tête *netpbm.h* en y déclarant et définissant la fonction *Netpbm_loadP6()* ayant le même prototype que la fonction *Netpbm_loadP3()*.

La définition de cette fonction doit commencer par ouvrir un flux en lecture binaire sur le fichier spécifié. Si cette opération échoue, il faut terminer la fonction en renvoyant un pointeur nul pour signaler l'échec. Vient ensuite la lecture du préambule textuel (c'est autorisé même sur un flux binaire) similaires au cas *P3* si ce n'est que le préfixe attendu sera *P6*[126]. Il faut toutefois s'assurer que la dernière ligne de texte ait été entièrement consommée avant d'envisager la lecture des données binaires. L'ajout de *"\n"* dans la chaîne de format à la suite de la position du troisième entier (celui devant valoir *255*) permet de consommer le retour à la ligne. Après l'allocation dynamique des *uint8_t* nécessaires à la mémorisation de cette image, leur lecture depuis le flux a lieu de manière binaire : un seul appel à *fread()* est suffisant. En cas d'échec, il faudra libérer le tableau obtenu dynamiquement, fermer le flux et terminer la fonction par le renvoi d'un pointeur nul. Il ne reste plus alors qu'à fermer le flux, mettre à jour les entiers du contexte appelant avec la largeur et la hauteur de l'image, avant de renvoyer le tableau décrivant son contenu.

Une fonction *test_loadP6()*, attendant en paramètres deux chaînes *inputFileName* et *outputFileName* doit être réalisée dans le fichier *prog04.c* et sera appelée par le programme principal en utilisant *"output_D.ppm"* et *"output_E.ppm"* comme paramètres effectifs. Après l'affichage de son nom et de ses paramètres, elle doit invoquer la fonction *Netpbm_loadP6()* afin d'obtenir les dimensions et le contenu de l'image indiquée par le paramètre *inputFileName*. En cas d'échec du chargement, il faut produire un message d'erreur sur *stderr* et quitter la fonction. Si l'image est chargée, ses dimensions devront être affichées dans le terminal et tous ses pixels devront être transformés par une permutation des composantes : bleu devient rouge, rouge devient vert, vert devient bleu.

Il ne reste plus qu'à sauvegarder cette image, sous le nom que *outputFileName* indique, par un appel à la fonction *Netpbm_saveP6()*, en produisant un message d'erreur sur *stderr* en cas d'échec, et à libérer la mémoire allouée dynamiquement (obtenue lors du chargement).

Fabriquez à nouveau le programme et exécutez-le. La visualisation du fichier *output_E.ppm* (en cliquant dessus par exemple) doit représenter une image dont les dimensions sont les mêmes que celles de *output_D.ppm* mais dont les dégradés de couleurs ont été permutés pour revenir à ceux de l'image *output_A.ppm*.

8.9. Analyse de la ligne de commande

À l'occasion de la présentation des chaînes de caractères, il a été mentionné en 5.4 que la fonction principale d'un programme pouvait recevoir en paramètres les chaînes qui constituent les mots de la ligne de commande. De plus, en 8.6 nous venons de découvrir que la fonction

126 En toute rigueur, le préambule peut contenir des commentaires que nous n'envisageons pas ici.

standard *sscanf()* permettait d'analyser le contenu d'une chaîne pour en extraire des valeurs. Nous réutilisons ici ces fonctionnalités afin de choisir, parmi tous les tests que nous venons de réaliser, ceux qui devront être invoqués.

Complétez alors le prototype de la fonction *main()* du fichier *prog04.c* pour que la ligne de commande lui soit accessible. Si aucun mot ne suit le nom du programme, il faut produire sur *stderr* un message "*usage: PPPP test_numbers...\n*" dans lequel *PPPP* doit être substitué par le nom du programme tel qu'il figure sur la ligne de commande, puis terminer immédiatement l'exécution du programme en renvoyant *1*.

Vient ensuite l'analyse de chaque mot de la ligne de commande (à l'exception du nom du programme). Si la fonction standard *strcmp()* indique qu'un de ces mots est exactement "*all*" il faudra invoquer chacun des dix tests réalisés. Si au contraire, ce mot permet l'extraction d'un entier à l'aide de la fonction standard *sscanf()*, il ne faudra invoquer que le test désigné (on les numérotera de *1* à *10* selon l'ordre dans lequel ils ont été réalisés). S'il est impossible d'obtenir un tel numéro ou si ce dernier ne désigne aucun des dix tests réalisés, il faudra signifier la situation sur *stderr* et passer à l'analyse du mot suivant.

(http://en.cppreference.com/w/c/language/main_function)

(http://en.cppreference.com/w/c/string/byte/strcmp)

(http://en.cppreference.com/w/c/io/fscanf)

Fabriquez à nouveau le programme et exécutez-le afin de vérifier qu'il respecte bien toutes les exigences décrites ici.

8.10. Résumé

Ici s'achève cette quatrième séance pratique dans laquelle nous nous sommes familiarisés avec les moyens d'entrées-sorties. Il s'agit principalement des fonctionnalités qui permettent de lire et d'écrire dans des fichiers. Toutefois, ce procédé est bien plus général puisque l'usage du terminal, que ce soit pour afficher des messages ou pour saisir au clavier, repose sur la même notion de flux.

Lorsque ces flux manipulent du texte, ils proposent des fonctionnalités de formatage permettant de transformer en texte des données de types variés et d'extraire d'un texte de telles valeurs. Ces fonctionnalités de formatage sont également utilisables sur des chaînes de caractères indépendantes de tout flux.

Les flux d'entrées-sorties offrent également la possibilité d'échanger des données binaires. Il s'agit de transférer par blocs des données telles qu'elles figurent dans la mémoire de la machine. Cela économise de nombreuses et coûteuses (en temps) opérations de conversions entre les données numériques et du texte lorsque la forme textuelle n'est pas strictement indispensable (l'utilisateur ne la lira pas).

Pour vous entraîner, et en vue de passer l'épreuve pratique, veuillez réaliser un programme qui manipule des structures de données (découvertes dans le cours du chapitre 7) pour les sauvegarder et les relire depuis des fichiers, sous forme textuelle ou binaire.

Voici le contenu des fichiers réalisés dans ce sujet, tels qu'ils doivent être si vous avez suivi scrupuleusement les recommandations qui vous ont été données à chaque étape[127].

```
//---- prog04.c ----
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "netpbm.h"

void
test_output(void)
{
printf("\n~~~~ %s() ~~~~\n",__func__);
int i=1234;
double d=12.34;
char c='A';
char s[]="hello";
char line[26]; // a bit too small
snprintf(line,sizeof(line),"i=%d d=%g c=%c s=%s",i,d,c,s);
printf("<%s>\n",line);
fprintf(stdout,"out: <%s>\n",line);
fprintf(stderr,"err: <%s>\n",line);
}

void
test_txtWrite(const char *fileName)
{
printf("\n~~~~ %s(%s) ~~~~\n",__func__,fileName);
FILE *f=fopen(fileName,"w");
if(!f)
  { fprintf(stderr,"cannot open %s\n",fileName); return; }
int i=1234;
double d=12.34;
char c='A';
char s[]="hello";
fprintf(f,"an integer: %d\n",i);
fprintf(f,"a real: %g\n",d);
fprintf(f,"a char: %c\n",c);
fprintf(f,"a string: %s\n",s);
fclose(f);
}

void
test_saveP3(const char *fileName,
            int width,
            int height)
{
printf("\n~~~~ %s(%s,%d,%d) ~~~~\n",__func__,fileName,width,height);
const int byteCount=3*width*height;
uint8_t *data=(uint8_t *)malloc((size_t)byteCount);
if(!data) { abort(); }
for(int y=0;y<height;++y)                                                    // ...(1/4)...
```

127 Les chaînes de caractères ayant été étudiées, il est désormais envisageable d'utiliser la variable `__func__` pour afficher le nom de chaque fonction de test. En effet, depuis la version C99, chaque fonction fournit implicitement (sans avoir à la déclarer) une chaîne de caractères constante nommée `__func__` qui en désigne le nom.

```
  {                                                                    // ...(2/4)...
  for(int x=0;x<width;++x)
    {
    const int red=255*x/(width-1);
    const int green=255*y/(height-1);
    const int blue=255-(red+green)/2;
    const int idx=3*(y*width+x);
    data[idx+0]=(uint8_t)red;
    data[idx+1]=(uint8_t)green;
    data[idx+2]=(uint8_t)blue;
    }
  }
if(!Netpbm_saveP3(fileName,width,height,data))
  { fprintf(stderr,"cannot save %s\n",fileName); }
free(data);
}

void
test_input(void)
{
printf("\n~~~~ %s() ~~~~\n",__func__);
char line[100], fileName[100]="";
int width=0, height=0;
do
  {
  printf("file name?\n");
  if(!fgets(line,sizeof(line),stdin)) { return; }
  } while(sscanf(line,"%s",fileName)!=1);
do
  {
  printf("image size (width x height)?\n");
  if(!fgets(line,sizeof(line),stdin)) { return; }
  } while(sscanf(line,"%d x%d",&width,&height)!=2);
if(fileName[0]&&(width>0)&&(height>0))
  { test_saveP3(fileName,width,height); }
}

void
test_loadP3(const char *inputFileName,
            const char *outputFileName)
{
printf("\n~~~~ %s(%s,%s) ~~~~\n",__func__,inputFileName,outputFileName);
int width,height;
uint8_t *data=Netpbm_loadP3(inputFileName,&width,&height);
if(!data)
  { fprintf(stderr,"cannot load %s\n",inputFileName); return; }
printf("width=%d height=%d\n",width,height);
int byteCount=3*width*height;
for(int i=0;i<byteCount;i+=3)
  {
  uint8_t red=data[i+0], green=data[i+1], blue=data[i+2];
  data[i+0]=blue; data[i+1]=red; data[i+2]=green;
  }
if(!Netpbm_saveP3(outputFileName,width,height,data))
  { fprintf(stderr,"cannot save %s\n",outputFileName); }
free(data);
}                                                                      // ...(2/4)...
```

```
void                                                                    // ...(3/4)...
test_binWrite(const char *fileName)
{
printf("\n~~~~ %s(%s) ~~~~\n",__func__,fileName);
int iArray[5]={ 10, 20, 30, 40, 50 };
double dArray[4]={ 10.01, 20.02, 30.03, 40.04 };
for(int i=0;i<5;++i) { printf("%d ",iArray[i]); } printf("\n");
for(int i=0;i<4;++i) { printf("%g ",dArray[i]); } printf("\n");
FILE *f=fopen(fileName,"wb");
if(!f)
  { fprintf(stderr,"cannot open %s\n",fileName); return; }
if(fwrite(iArray,sizeof(int),5,f)!=5)
  { fprintf(stderr,"cannot write to %s\n",fileName); fclose(f); return; }
if(fwrite(dArray,sizeof(double),4,f)!=4)
  { fprintf(stderr,"cannot write to %s\n",fileName); fclose(f); return; }
fclose(f);
}

void
test_binRead(const char *fileName)
{
printf("\n~~~~ %s(%s) ~~~~\n",__func__,fileName);
int iArray[5];
double dArray[4];
FILE *f=fopen(fileName,"rb");
if(!f)
  { fprintf(stderr,"cannot open %s\n",fileName); return; }
if(fread(iArray,sizeof(int),5,f)!=5)
  { fprintf(stderr,"cannot read from %s\n",fileName); fclose(f); return; }
if(fread(dArray,sizeof(double),4,f)!=4)
  { fprintf(stderr,"cannot read from %s\n",fileName); fclose(f); return; }
fclose(f);
for(int i=0;i<5;++i) { printf("%d ",iArray[i]); } printf("\n");
for(int i=0;i<4;++i) { printf("%g ",dArray[i]); } printf("\n");
}

void
test_saveP6(const char *inputFileName,
            const char *outputFileName)
{
printf("\n~~~~ %s(%s,%s) ~~~~\n",__func__,inputFileName,outputFileName);
int width,height;
uint8_t *data=Netpbm_loadP3(inputFileName,&width,&height);
if(!data)
  { fprintf(stderr,"cannot load %s\n",inputFileName); return; }
printf("width=%d height=%d\n",width,height);
int byteCount=3*width*height;
for(int i=0;i<byteCount;i+=3)
  {
  uint8_t red=data[i+0], green=data[i+1], blue=data[i+2];
  data[i+0]=green; data[i+1]=blue; data[i+2]=red;
  }
if(!Netpbm_saveP6(outputFileName,width,height,data))
  { fprintf(stderr,"cannot save %s\n",outputFileName); }
free(data);
}
                                                                        // ...(3/4)...
```

```
void                                                                    // ...(4/4)...
test_loadP6(const char *inputFileName,
            const char *outputFileName)
{
printf("\n~~~~ %s(%s,%s) ~~~~\n",__func__,inputFileName,outputFileName);
int width,height;
uint8_t *data=Netpbm_loadP6(inputFileName,&width,&height);
if(!data)
  { fprintf(stderr,"cannot load %s\n",inputFileName); return; }
printf("width=%d height=%d\n",width,height);
int byteCount=3*width*height;
for(int i=0;i<byteCount;i+=3)
  {
  uint8_t red=data[i+0], green=data[i+1], blue=data[i+2];
  data[i+0]=blue; data[i+1]=red; data[i+2]=green;
  }
if(!Netpbm_saveP6(outputFileName,width,height,data))
  { fprintf(stderr,"cannot save %s\n",outputFileName); }
free(data);
}

int
main(int argc,
     char **argv)
{
if(argc<2)
  { fprintf(stderr,"usage: %s test_numbers...\n",argv[0]); return 1; }
for(int i=1;i<argc;++i)
  {
  int minTest=1, maxTest=10;
  if(strcmp(argv[i],"all"))
    {
    if(sscanf(argv[i],"%d",&minTest)==1) { maxTest=minTest; }
    else { fprintf(stderr,"ignoring %s\n",argv[i]); continue; }
    }
  for(int test=minTest;test<=maxTest;++test)
    {
    printf("\n#### test %d ####",test);
    switch(test)
      {
      case  1: { test_output();                                    break; }
      case  2: { test_txtWrite("output_text.txt");                 break; }
      case  3: { test_txtWrite("/output_forbidden.txt");           break; }
      case  4: { test_saveP3("output_A.ppm",320,200);              break; }
      case  5: { test_input();                                     break; }
      case  6: { test_loadP3("output_A.ppm","output_C.ppm");       break; }
      case  7: { test_binWrite("output_data.bin");                 break; }
      case  8: { test_binRead("output_data.bin");                  break; }
      case  9: { test_saveP6("output_A.ppm","output_D.ppm");       break; }
      case 10: { test_loadP6("output_D.ppm","output_E.ppm");       break; }
      default: { fprintf(stderr,"unexpected test number %d\n",test); break; }
      }
    }
  }
return 0;
}
```

```
//---- netpbm.h ----
#ifndef NETPBM_H
#define NETPBM_H 1

#include <stdbool.h>
#include <stdint.h>

bool // success
Netpbm_saveP3(const char *fileName,
              int width,
              int height,
              uint8_t *data);

uint8_t * // loaded data or NULL
Netpbm_loadP3(const char *fileName,
              int *out_width,
              int *out_height);

bool // success
Netpbm_saveP6(const char *fileName,
              int width,
              int height,
              uint8_t *data);

uint8_t * // loaded data or NULL
Netpbm_loadP6(const char *fileName,
              int *out_width,
              int *out_height);

#endif // NETPBM_H
```

```
//---- netpbm.c ----

#include "netpbm.h"
#include <stdio.h>
#include <stdlib.h>

bool // success
Netpbm_saveP3(const char *fileName,
              int width,
              int height,
              uint8_t *data)
{
FILE *f=fopen(fileName,"w");
if(!f) { return false; }
fprintf(f,"P3\n%d %d\n255\n",width,height);
const int byteCount=3*width*height;
for(int i=0;i<byteCount;++i)
  { fprintf(f,"%d%c",data[i],(i%18)==17 ? '\n' : ' '); }
if(byteCount%17) { fprintf(f,"\n"); }
fclose(f);
return true;
}

                                                                    // ...(1/3)...
```

```
                                                                    // ...(2/3)...
uint8_t * // loaded data or NULL
Netpbm_loadP3(const char *fileName,
              int *out_width,
              int *out_height)
{
FILE *f=fopen(fileName,"r");
if(!f) { return NULL; }
int width,height,range;
if((fgetc(f)!='P')||(fgetc(f)!='3')||
   (fscanf(f,"%d %d %d",&width,&height,&range)!=3)||
   (width<=0)||(height<=0)||(range!=255))
  { fclose(f); return NULL; }
int byteCount=3*width*height;
uint8_t *data=(uint8_t *)malloc((size_t)byteCount);
if(!data) { abort(); }
for(int i=0;i<byteCount;++i)
  {
#if 0
  int value;
  if(fscanf(f,"%d",&value)!=1)
    { free(data); fclose(f); return NULL; }
  data[i]=(uint8_t)value;
#else
  if(fscanf(f,"%hhu",data+i)!=1)
    { free(data); fclose(f); return NULL; }
#endif
  }
fclose(f);
//-- store out-parameters --
*out_width=width;
*out_height=height;
return data;
}

bool // success
Netpbm_saveP6(const char *fileName,
              int width,
              int height,
              uint8_t *data)
{
FILE *f=fopen(fileName,"wb");
if(!f) { return false; }
fprintf(f,"P6\n%d %d\n255\n",width,height);
const int byteCount=3*width*height;
fwrite(data,(size_t)byteCount,1,f);
fclose(f);
return true;
}









                                                                    // ...(2/3)...
```

```
                                                                    // ...(3/3)...

uint8_t * // loaded data or NULL
Netpbm_loadP6(const char *fileName,
              int *out_width,
              int *out_height)
{
FILE *f=fopen(fileName,"rb");
if(!f) { return NULL; }
int width,height,range;
if((fgetc(f)!='P')||(fgetc(f)!='6')||
   (fscanf(f,"%d %d %d\n",&width,&height,&range)!=3)||
   (width<=0)||(height<=0)||(range!=255))
  { fclose(f); return NULL; }
int byteCount=3*width*height;
uint8_t *data=(uint8_t *)malloc((size_t)byteCount);
if(!data) { abort(); }
if(fread(data,(size_t)byteCount,1,f)!=1)
  { free(data); fclose(f); return NULL; }
fclose(f);
//-- store out-parameters --
*out_width=width;
*out_height=height;
return data;
}
```

9. L05_Bitwise : Opérations bit-à-bit

Motivation :

Il existe des contextes applicatifs dans lesquels les entiers manipulés ne sont pas considérés pour la valeur arithmétique qu'ils représentent mais au contraire pour chacun de leurs *bits* constitutifs pris isolément (ou par sous-ensembles). C'est le cas notamment lorsqu'il est question de faire interagir le programme avec des composants matériels pour lesquels chaque *bit* représente une consigne ou un état (ceci sera étudié dans les matières S5-MIP et S6-MIP de l'ENIB).

Le langage C introduit à ce propos un jeu d'opérations ***bit-à-bit*** (*bitwise*, vocabulaire) qui vient compléter le jeu d'opérations arithmétiques usuelles et qui s'utilisent de la même façon. Ces opérations permettent de contrôler très efficacement les *bits* constitutifs des entiers.

(http://en.cppreference.com/w/c/language/operator_arithmetic)

Cette étude permettra également de constater la représentation binaire utilisée sur les entiers signés et non-signés.

Consignes :

Au delà de la découverte très guidée de ces quelques nouvelles fonctionnalités, le travail demandé ici nécessitera d'être autonome dans la réutilisation des notions déjà acquises (tableaux, passages de paramètres...) afin de les appliquer à un cas d'usage concret. Il faudra également savoir consulter la documentation. Veilliez à suivre scrupuleusement chacune des étapes en vous assurant à chaque fois d'avoir bien observé et compris ce qui vous est décrit.

9.1. Mise en place du programme

Ouvrez un terminal et grâce à la ligne de commande du *shell*, placez vous dans le répertoire de travail choisi pour ce sujet. Saisissez par exemple :

```
$ mkdir S3PRC_L05_Bitwise ↲
$ cd S3PRC_L05_Bitwise ↲
```

Placez dans ce répertoire le fichier `GNUmakefile` générique mis à votre disposition.

(http://www.enib.fr/~harrouet/Data/Courses/GNUmakefile)

Nous suivrons en effet la démarche de programmation modulaire présentée au chapitre 2.

Le programme principal sera rédigé dans le fichier `prog05.c` et appellera une fonction `test_shift()` qui sera réalisée dans ce même fichier. Cette dernière qui, n'attend aucun paramètre et ne renvoie aucun résultat, se contentera pour l'instant d'afficher son nom.

Fabriquez ce programme et exécutez-le afin de vérifier que tout est en place. En l'état, il ne fait qu'afficher le nom de la fonction `test_shift()` :

```
$ make ↲
$ ./prog05 ↲
```

9.2. Les opérations de décalage

Si nous nous représentons le motif de *bits* constitutif d'un entier comme une séquence allant de la gauche (les *bits* de poids forts) vers la droite (les *bits* de poids faibles), une première intervention envisageable consiste à décaler cette séquence de quelques positions vers la gauche ou la droite. Nous obtenons alors un motif semblable mais avec des bits supprimés dans la direction du décalage et un nombre équivalent de bits insérés du côté opposé.

Les expressions `expr<<N` et `expr>>N` provoquent respectivement le décalage de `N` *bits* vers la gauche (les poids forts) et vers la droite (les poids faibles) de l'expression `expr`[128].

Pour expérimenter cela, il faut compléter la fonction `test_shift()` de cette façon :

```
uint32_t values[]={ 12, 23, 34, 45, 56 };
const int count=(int)(sizeof(values)/sizeof(values[0]));
for(int i=0;i<count;++i)
  {
  const uint32_t value=values[i];
  for(int shift=0;shift<4;++shift)
    {
    const uint32_t ls=value<<shift, rs=value>>shift;
    printf("%u << %d = %u\t%u >> %d = %u\n",value,shift,ls,value,shift,rs);
    }
  }
```

Remarquez que pour l'instant nous ne nous intéressons qu'à des entiers non-signés (`uint32_t` ici) ; la cas des entiers signés sera étudié en 9.6. Pour les afficher fidèlement, même quand ils deviendront très grands, le format `%u` est utilisé avec la fonction standard `printf()`. Il s'agit de visualiser le résultat de quelques décalages à gauche et à droite sur quelques valeurs. Le type du résultat d'un tel décalage est celui de la valeur décalée ; la promotion vers le type `int` (éventuellement `unsigned`) s'applique toutefois pour les entiers de plus petit taille (voir en 4.4.3).

Relancez la fabrication du programme exécutable et exécutez-le afin d'observer les résultats. Vous devriez constater qu'un décalage nul est sans effet et que les autres décalages sont assimilables à des multiplications ou des divisions entières par des puissances de deux. En effet, décaler l'écriture d'un nombre exprimé dans une base de numération revient à multiplier ou diviser ce nombre par une puissance de la base choisie ; pensez à la représentation en base dix pour vous en convaincre.

Réalisez un nouveau module constitué uniquement du fichier d'en-tête `word32.h` pour l'instant ; il sera dédié à quelques opérations classiques sur des mots de 32 *bits* (`uint32_t`). La plupart de ces opérations ne seront constituées que de très peu d'instructions, elles pourront par conséquent être qualifiées d'`inline static` (voir en 7.2.3) et donc être directement définies dans ce fichier d'en-tête.

Dans un premier temps, ce module doit simplement fournir une fonction `Word32_one()` attendant un entier en paramètre et renvoyant un `uint32_t`. Son rôle est de fournir un motif de 32 *bits* dont seul celui qui est désigné par le paramètre vaudra `1` et tous les autres vaudront `0`. Par convention, les programmeurs numérotent les *bits* en commençant à `0` au niveau de celui ayant le poids le plus faible. Ainsi, si cette fonction est appelée avec le paramètre `0`, le résultat sera `1`, le paramètre `1` produira `2`, `2` produira `4`, `3` produira `8` et ainsi de suite jusqu'à un paramètre valant au plus `31`. Il suffit donc, pour produire ce résultat, de partir d'un motif de *bits* valant `1` et de le décaler vers la gauche selon le paramètre transmis.

Toutefois, un compilateur rigoureux devrait indiquer qu'une conversion d'un entier signé vers un entier non-signé risque d'entraîner une perte d'information. En effet, la constante littérale `1` est signée donc son décalage le sera aussi. Pour spécifier au compilateur que nous partons

128 Exactement comme `expr+N` représente une addition.

d'une valeur littérale non-signée il suffit de lui ajouter le suffixe `u` : la constante littérale `1u` est ainsi non-signée et son décalage le restera.

Complétez maintenant la fonction `test_shift()` pour afficher le résultat de cette nouvelle fonction pour tous les bits de `0` à `31`. L'affichage doit toujours utiliser le format `%u` afin que tous les résultats soient correctement représentés. Après une nouvelle compilation et la relance du programme exécutable, vous devriez voir apparaître toutes les puissances de deux correspondantes.

Pour terminer avec ces opérations de décalage, veuillez noter qu'il est possible de les combiner à une opération d'affectation comme nous le faisons déjà pour les opérations arithmétiques usuelles. Ainsi, les expressions `a<<=b` et `c>>=d` sont respectivement équivalentes à `a=a<<b` et `c=c>>d`.

9.3. Les combinaisons bit-à-bit

De la même façon que les opérations arithmétiques usuelles combinent les motifs de *bits* de deux opérandes pour produire le motif de *bits* du résultat, selon les règles de l'addition par exemple, il existe des opérateurs *bit-à-bit* qui combinent ces motifs de *bits* selon les règles des opérations logiques élémentaires.

Ces opérateurs sont `&` pour l'opération *et*, `|` pour l'opération *ou* et `^` pour l'opération *ou-exclusif*.

```
  c=a&b; // bitwise AND              c=a|b; // bitwise OR               c=a^b; // bitwise X-OR

  a7 a6 a5 a4 a3 a2 a1 a0            a7 a6 a5 a4 a3 a2 a1 a0            a7 a6 a5 a4 a3 a2 a1 a0
& b7 b6 b5 b4 b3 b2 b1 b0          | b7 b6 b5 b4 b3 b2 b1 b0          ^ b7 b6 b5 b4 b3 b2 b1 b0
  ————————————————————————           ————————————————————————           ————————————————————————
  c7 c6 c5 c4 c3 c2 c1 c0            c7 c6 c5 c4 c3 c2 c1 c0            c7 c6 c5 c4 c3 c2 c1 c0
  ( avec ci = ai & bi )              ( avec ci = ai | bi )              ( avec ci = ai ^ bi )
```

Comme explicité sur cette illustration (qui se limite à des entiers de huit *bits* pour réduire l'écriture), ces opérations consistent à appliquer individuellement l'opération logique sur chaque paire de *bits* qui sont en même position dans chacun des deux opérandes.

Cela est très différent des opérateurs logiques du langage C qui ne dépendent que de l'éventuelle nullité de ses opérandes :

- l'opération logique `c=a&&b;` est équivalente à `c=(a!=0)&&(b!=0) ? true : false;` alors que l'opération *bit-à-bit* `c=a&b;` peut produire toute une variété de valeurs,
- l'opération logique `c=a||b;` est équivalente à `c=(a!=0)||(b!=0) ? true : false;` alors que l'opération *bit-à-bit* `c=a|b;` peut produire toute une variété de valeurs.

Pour expérimenter ces opérations, faites en sorte que la fonction principale invoque une nouvelle fonction `test_bitwise()` qui affiche son nom et exécute cet algorithme :

```
uint32_t values[]={ 12, 23, 34, 45, 56 };
const int count=(int)(sizeof(values)/sizeof(values[0]));
for(int i=0;i<count;++i)
  {
  const uint32_t value=values[i];
  const uint32_t prev=i ? values[i-1] : 0;
  printf("  %u & %u = %u\n",value,prev,value&prev);
  printf("  %u | %u = %u\n",value,prev,value|prev);
  printf("  %u ^ %u = %u\n",value,prev,value^prev);
  }
```

Après avoir fabriqué à nouveau le programme exécutable, son exécution vous fera constater la variété de valeurs produites par ces opérations.

Afin de mieux comprendre les valeurs affichées précédemment, nous devons ajouter une nouvelle fonction `Word32_test()` au fichier d'en-tête `word32.h`. Elle sera également qualifiée de `inline static` et devra renvoyer un booléen indiquant si la valeur de type `uint32_t` passée en argument a son *bit*, dont le numéro est spécifié par un autre argument entier, qui vaut `1`. Il suffit pour ceci de fabriquer un ***masque binaire*** (vocabulaire) avec ce seul *bit* positionné (grâce à la fonction `Word32_one()`) et de le combiner selon l'opérateur `&` avec la valeur transmise. Partout où le masque a ses *bits* nuls, le résultat aura aussi des *bits* nuls (en logique, `0` et `x` vaut `0`, quel que soit `x`) ; au contraire, le seul emplacement du masque pour lequel le *bit* n'est pas nul conduira à une recopie du *bit* correspondant de la valeur d'entrée dans le résultat (en logique, `1` et `x` vaut `x`). Le résultat sera donc entièrement nul si le *bit* en question était nul dans la valeur d'entrée, et sera non nul si le *bit* en question était non nul ; l'interprétation booléenne de ce résultat renseigne donc sur l'état du *bit* choisi dans la valeur d'entrée.

Il est désormais possible de faire précéder, dans la fonction `test_bitwise()`, l'affichage des calculs par une boucle qui, grâce la fonction `Word32_test()`, teste un à un tous les *bits* de la valeur courante et n'affiche la position que de ceux qui sont non nuls. Ainsi, après une nouvelle compilation et une relance du programme exécutable, l'énumération des *bits* présents dans le valeurs combinées vous aidera à interpréter les résultats des opérations préalablement observées.

Pour terminer avec ces opérations *bit-à-bit*, veuillez noter qu'il est possible de les combiner à une opération d'affectation comme nous le faisons déjà pour les opérations arithmétiques usuelles. Ainsi, les expressions `a&=b`, `c|=d` et `e^=f` sont respectivement équivalentes à `a=a&b`, `c=c|d` et `e=e^f`.

9.4. Test de bits et forçage à 1

La fonction `Word32_test()`, qui permet de tester l'état d'un *bit* particulier, a déjà été réalisée. Nous la réutilisons ici dans une nouvelle fonction `Word32_dump()` qui a pour rôle l'inscription dans une chaîne de trente-deux caractères passée en paramètre, de l'état des *bits* d'un autre paramètre de type `uint32_t`. Il suffit pour cela de passer en revue les trente-deux *bits* de la valeur en question, à l'aide de la fonction `Word32_test()`, pour inscrire le caractère littéral `'.'` ou `'X'` selon l'état du *bit* concerné (`0` ou `1`). Pour une lecture facilitée, les *bits* de poids forts seront représentés en début de chaîne et les poids faibles en fin. Cette nouvelle fonction ne se limite pas à quelques instructions et mérite d'être définie de manière conventionnelle dans un fichier de définition `word32.c`.

Pour tester cette fonction, il faudra réaliser dans le fichier `prog05.c` une fonction `test_dump_load()` qui affichera son nom et sera appelée par la fonction principale. Elle devra disposer d'un tableau de trente-trois caractères : un pour chaque *bit* à représenter et le zéro terminal. Affichez alors ce que produit `Word32_dump()` pour quelques dizaines de valeurs.

Fabriquez à nouveau ce programme et exécutez-le afin de vérifier que le motif produit correspond bien aux valeurs transmises.

Nous envisageons maintenant la définition `inline static`, dans le fichier `word32.h`, d'une fonction `Word32_set()` dont le rôle est de forcer à `1` un *bit* choisi dans un entier de type `uint32_t` du contexte appelant. Il suffit pour ceci de fabriquer un masque binaire avec ce seul *bit* positionné (grâce à la fonction `Word32_one()`) et de le combiner selon l'opérateur `|` avec la valeur à modifier. Partout où le masque a ses *bits* nuls, les *bits* de la valeur seront inchangés (en logique, `0` ou `x` vaut `x`) ; au contraire, le seul emplacement du masque pour lequel le *bit* n'est pas nul forcera à `1` le *bit* de la valeur à modifier (en logique, `1` ou `x` vaut `1`, quel que soit `x`).

Cette nouvelle fonction peut désormais être exploitée dans une fonction `Word32_load()` de `word32.c`. Son rôle est complémentaire à `Word32_dump()` : elle analyse la chaîne de trente-deux caractères qui lui est transmise afin de constituer un entier de type `uint32_t` qu'elle

renverra. Il suffit pour ceci de partir d'une valeur nulle et de forcer à `1`, à l'aide de la fonction `Word32_set()`, les *bits* de cette valeur lorsqu'un caractère vaut `'X'` (en considérant que la chaîne commence par les poids forts).

Complétez alors la boucle de la fonction `test_load_save()` pour vérifier que l'analyse de la chaîne de caractères par la fonction `Word32_load()` fournit bien la valeur qui avait été transmise à la fonction `Word32_dump()` lorsqu'elle a produit cette chaîne.

Fabriquez à nouveau ce programme et exécutez-le afin de vérifier que nous obtenons bien le résultat attendu.

9.5. Complémentation et forçage de bits à 0

De la même façon qu'en arithmétique et en logique il existe des opérateurs unaires pour transformer un unique opérande (`-x` en arithmétique produit l'opposé de la valeur `x`, `!c` en logique produit la négation de la condition `c`), les opérations *bit-à-bit* fournissent un opérateur de complémentation `~` dont le rôle est d'inverser l'état de chaque *bit* de l'entier considéré : les `0` deviennent des `1` et les `1` deviennent des `0`.

Cela est très différent de la négation logique qui ne dépend que de l'éventuelle nullité de son opérande : la négation logique `b=!a;` est équivalente à `b=(a==0) ? true : false;` alors que le complément *bit-à-bit* `b=~a;` peut produire toute une variété de valeurs.

Pour constater l'effet de cet opérateur, il faudra réaliser dans le fichier `prog05.c` une fonction `test_complement()` qui affichera son nom et sera appelée par la fonction principale. Elle contiendra une première variable `v1` de type `uint32_t` ayant une valeur quelconque et une seconde variable `v2` valant le complément de la première. La fonction `Word32_dump()` permettra de visualiser facilement le motif de *bits* de ces deux valeurs.

Fabriquez à nouveau ce programme et exécutez-le afin de vérifier que les motifs produits sont bien complémentaires.

Nous réutilisons cette nouvelle connaissance pour compléter le fichier `word32.h` avec une fonction `Word32_clear()` qui sera qualifiée de `inline static`. Son rôle sera de forcer à `0` un *bit* choisi dans un entier de type `uint32_t` du contexte appelant. Il suffit pour ceci de fabriquer un masque binaire avec ce seul *bit* positionné (grâce à la fonction `Word32_one()`), de lui appliquer la complémentation afin qu'il soit désormais constitué d'un unique *bit* nul et de le combiner selon l'opérateur `&` avec la valeur à modifier. Partout où le masque a ses *bits* non nuls, les *bits* de la valeur seront inchangés (en logique, `1` et `x` vaut `x`) ; au contraire, le seul emplacement du masque pour lequel le *bit* est nul forcera à `0` le *bit* de la valeur à modifier (en logique, `0` et `x` vaut `0`, quel que soit `x`).

Pour tester le bon fonctionnement de cette nouvelle fonctionnalité, nous compléterons la fonction `test_complement()` afin de fabriquer une variable `v3` qui sera également le complément de `v1` mais en forçant ses *bits* à `0`. Il faudra donc donner à `v3` une valeur initiale constituée exclusivement de *bits* valant `1`. Un tel motif s'obtient simplement en utilisant le complément de la constante littérale `0`. Seulement, comme déjà vu en 9.2, un compilateur rigoureux devrait indiquer qu'une conversion d'un entier signé vers un entier non-signé risque d'entraîner une perte d'information. En effet, la constante littérale `0` est signée donc son complément le sera aussi. Pour spécifier au compilateur que nous partons d'une valeur littérale non-signée il suffit de lui ajouter le suffixe `u` : la constante littérale `0u` est ainsi non-signée et son complément le restera. Ensuite, pour chaque *bit* de `v1`, lorsque la fonction `Word32_test()` indique la valeur `true`, il faudra utiliser la fonction `Word32_clear()` pour forcer à `0` le *bit* correspondant dans `v3`. La fonction `Word32_dump()` servira encore à rendre explicite le résultat de ce traitement.

Fabriquez à nouveau ce programme et exécutez-le afin de vérifier que le motif ainsi produit pour `v3` est bien identique à celui obtenu sur `v2` par complémentation de `v1`.

9.6. Opérations bit-à-bit sur les entiers signés

De manière générale, il est très rare que des opérations bit-à-bit soient nécessaires sur des entiers signés. Si le signe a effectivement une importance, c'est que la valeur doit être

interprétée d'un point de vue arithmétique en ne doit pas être considérée comme une simple séquence de *bits* à manipuler individuellement. C'est pour cette raison que nous avons utilisé le type non-signé `uint32_t` dans nos investigations autour de ce thème. Néanmoins, les opérations bit-à-bit sont quand même autorisées sur les entiers signés et fonctionnent de la même façon à une exception près : le décalage vers la droite (vers les poids faibles) nécessite une attention particulière.

Pour aborder ce point particulier il est nécessaire de comprendre la représentation qui est utilisée pour les entiers signés. À cette fin, nous réaliserons dans le fichier `prog05.c` une nouvelle fonction `test_signed()` qui affichera son nom et sera appelée par la fonction principale. Elle utilisera dans un premier temps trois variables `v1`, `v2` et `v3` de type `int32_t` ; la première recevra une valeur positive quelconque, la deuxième son opposé et la troisième son complément. Pour chacune de ces trois variables, il faudra visualiser le motif de *bits* avec la fonction `Word32_dump()` (un *cast* est nécessaire pour le paramètre non-signé) et afficher la valeur (avec le format `%d` cette fois).

Fabriquez à nouveau ce programme et exécutez-le afin d'observer le motif produit et la valeur entière pour chacune des trois variables. Vous devriez constater que `v2`, qui contient une valeur négative, a ses `bits` de poids forts positionnés, tout comme `v3` qui contient le complément d'une valeur positive. Les motifs de `v2` et `v3` sont même très similaires et la valeur de `v3` est juste inférieure d'une unité à celle de `v2`. Nous en déduisons que nous pouvons émuler l'opposé d'une valeur entière en ajoutant `1` à son complément : il s'agit de la représentation en ***complément-à-deux*** (vocabulaire). L'intérêt de cette solution tient dans le fait que les opérations arithmétiques usuelles sont réalisées exactement de la même façon, que les valeurs entières soient positives ou négatives.

(https://en.wikipedia.org/wiki/Two%27s_complement)

Pour vérifier ce constat, ajoutons au fichier `word32.h` une nouvelle fonction `Word32_negate()` qui sera qualifiée de `inline static`. Son rôle est d'émuler la négation de son paramètre de type `uint32_t` en renvoyant son complément-à-deux. Complétez alors la fonction `test_signed()` en ajoutant une variable `v4` de type `int32_t` initialisée avec le résultat de `Word32_negate()` appliquée à `v1` (des *cast*s sont nécessaires pour le paramètre et le résultat) et en affichant également son motif de *bits* et sa valeur.

Fabriquez à nouveau ce programme et exécutez-le afin de comparer `v4` à `v2`.

En 9.2 nous constations que les opérations de décalage pouvaient s'interpréter du point de vue arithmétique comme des multiplications ou des divisions entières par des puissances de deux. Multiplier ou diviser par une quantité positive ne doit pas changer le signe ; il faut donc que les opérations de décalage respectent cette contrainte sur les entiers signés.

Nous avons également constaté que la représentation en complément-à-deux des valeurs négatives positionnait les *bits* de poids forts. Puisque le décalage vers la droite (les poids faibles) nécessite l'injection de *bits* au niveau des poids forts, il convient donc de les choisir avec précaution afin de respecter le signe du résultat.

Expérimentons en ajoutant dans la fonction `test_signed()` une variable `v5`, de type `int32_t`, valant `v1` (positive) décalée de quelques *bits* vers la droite, et visualisons son motif de *bits* et sa valeur. Après compilation et relance du programme nous constatons un comportement similaire à ce que nous avons vu en 9.2 : des *bits* nuls ont été injectés au niveau des poids forts.

Poursuivons la même démarche avec une variable `v6`, toujours de type `int32_t`, valant `v2` (négative) décalée de quelques *bits* vers la droite. Après compilation et relance du programme nous constatons un comportement différent : des *bits* non nuls ont été injectés au niveau des poids forts, ce qui permet au résultat de rester négatif et conforme à l'arithmétique.

Comparons ce comportement avec celui d'une variable `v7`, de type `uint32_t` cette fois, valant `v2` (négative) convertie en `uint32_t` (rendue non-signée donc) et décalée de quelques *bits* vers la droite. Après compilation et relance du programme nous constatons un comportement

similaire au cas positif : des *bits* nuls ont été injectés au niveau des poids forts et l'affichage sous forme d'entier signé n'est alors plus conforme aux attentes.

Nous venons de vérifier que le décalage vers la droite implique la propagation du *bit* de poids fort (injection de *bits* identiques au *bit* de poids fort initial) dans le cas d'un entier signé alors que des `0` sont systématiquement utilisés dans le cas d'un entier non-signé.

9.7. Résumé

Ici s'achève cette cinquième séance pratique dans laquelle nous avons découvert et expérimenté les opérations *bit-à-bit*. Nous avons constaté qu'il ne s'agit ni plus ni moins que d'opérateurs dont l'utilisation est semblable à celle des opérateurs arithmétiques. Ceux d'entre eux qui attendent deux opérandes peuvent même être combinés avec l'opération d'affectation (`<<=`, `>>=`, `&=`, `|=`, `^=`) tout comme dans le cas des opérations arithmétiques usuelles (`+=`, `-=`, `*=`, `/=`, `%=`).

Ces opérations s'appliquent à des entiers mais ne traitent pas directement de la signification arithmétique de leur valeur ; elles les considèrent plutôt comme des séquences de *bits* auxquels on applique des opérations logiques élémentaires. Ces opérations concernent les décalages des motifs de *bits* vers les poids forts (`<<`) ou les poids faibles (`>>`), leur combinaison selon des opérations *et* (`&`), *ou* (`|`) ou *ou-exclusif* (`^`) ou encore leur complémentation (`~`).

Les usages habituels de ces opérations consistent à tester l'état de quelques *bits* particuliers d'un entier, à les forcer à `1` ou à `0` ; ceci est particulièrement utile dans le cadre de programmes qui interagissent avec des composants matériels.

Voici le contenu des fichiers réalisés dans ce sujet, tels qu'ils doivent être si vous avez suivi scrupuleusement les recommandations qui vous ont été données à chaque étape.

```
//---- prog05.c ----
#include <stdio.h>
#include "word32.h"

void
test_shift(void)
{
printf("\n~~~~ %s() ~~~~\n",__func__);
uint32_t values[]={ 12, 23, 34, 45, 56 };
const int count=(int)(sizeof(values)/sizeof(values[0]));
for(int i=0;i<count;++i)
  {
  const uint32_t value=values[i];
  for(int shift=0;shift<4;++shift)
    {
    const uint32_t ls=value<<shift, rs=value>>shift;
    printf("%u << %d = %u\t%u >> %d = %u\n",value,shift,ls,value,shift,rs);
    }
  }
for(int i=0;i<32;++i)
  { printf("bit %d --> %u\n",i,Word32_one(i)); }
}

void
test_bitwise(void)
{
printf("\n~~~~ %s() ~~~~\n",__func__);
uint32_t values[]={ 12, 23, 34, 45, 56 };
const int count=(int)(sizeof(values)/sizeof(values[0]));
for(int i=0;i<count;++i)
  {
  const uint32_t value=values[i];
  printf("%u: ",value);
  for(int j=0;j<32;++j)
    {
    if(Word32_test(value,j))
      { printf("%u ",j); }
    }
  printf("\n");
  const uint32_t prev=i ? values[i-1] : 0;
  printf("  %u & %u = %u\n",value,prev,value&prev);
  printf("  %u | %u = %u\n",value,prev,value|prev);
  printf("  %u ^ %u = %u\n",value,prev,value^prev);
  }
}

void
test_dump_load(void)
{
printf("\n~~~~ %s() ~~~~\n",__func__);
char pattern[33]={0};
for(uint32_t i=0;i<40;++i)                                                    // ...(1/2)...
```

```
  {                                                                                    // ...(2/2)...
  Word32_dump(pattern,i);
  uint32_t check=Word32_load(pattern);
  printf("%s\t%u\t%u\n",pattern,i,check);
  }
}

void
test_complement(void)
{
printf("\n~~~~ %s() ~~~~\n",__func__);
char pattern[33]={0};
const uint32_t v1=9876, v2=~v1;
Word32_dump(pattern,v1); printf("v1: %s\t%u\n",pattern,v1);
Word32_dump(pattern,v2); printf("v2: %s\t%u\n",pattern,v2);
uint32_t v3=~0u;
for(int i=0;i<32;++i)
  {
  if(Word32_test(v1,i))
    { Word32_clear(&v3,i); }
  }
Word32_dump(pattern,v2); printf("v3: %s\t%u\n",pattern,v3);
}

void
test_signed(void)
{
printf("\n~~~~ %s() ~~~~\n",__func__);
char pattern[33]={0};
const int32_t v1=9876, v2=-v1, v3=~v1;
Word32_dump(pattern,(uint32_t)v1); printf("v1: %s\t%d\n",pattern,v1);
Word32_dump(pattern,(uint32_t)v2); printf("v2: %s\t%d\n",pattern,v2);
Word32_dump(pattern,(uint32_t)v3); printf("v3: %s\t%d\n",pattern,v3);
const int32_t v4=(int32_t)Word32_negate((uint32_t)v1);
Word32_dump(pattern,(uint32_t)v4); printf("v4: %s\t%d\n",pattern,v4);
const int32_t v5=v1>>2, v6=v2>>2;
Word32_dump(pattern,(uint32_t)v5); printf("v5: %s\t%d\n",pattern,v5);
Word32_dump(pattern,(uint32_t)v6); printf("v6: %s\t%d\n",pattern,v6);
const uint32_t v7=((uint32_t)v2)>>2;
Word32_dump(pattern,v7); printf("v7: %s\t%d\n",pattern,v7);
}

int
main(void)
{
test_shift();
test_bitwise();
test_dump_load();
test_complement();
test_signed();
return 0;
}
```

```
//---- word32.h ----
#ifndef WORD32_H
#define WORD32_H 1

#include <stdint.h>
#include <stdbool.h>

#define BIT_ZERO_CHAR '.'
#define BIT_ONE_CHAR  'X'

inline static
uint32_t
Word32_one(int bit)
{ return 1u<<bit; }

inline static
bool
Word32_test(uint32_t value,
            int bit)
{
const uint32_t mask=Word32_one(bit);
return value&mask;
}

inline static
void
Word32_set(uint32_t *inout_value,
           int bit)
{
const uint32_t mask=Word32_one(bit);
*inout_value|=mask;
}

inline static
void
Word32_clear(uint32_t *inout_value,
             int bit)
{
const uint32_t mask=Word32_one(bit);
*inout_value&=~mask;
}

inline static
uint32_t
Word32_negate(uint32_t value)
{ return 1+~value; }

void
Word32_dump(char *pattern,
            uint32_t value);

uint32_t
Word32_load(const char *pattern);

#endif // WORD32_H
```

```
//---- word32.c ----
#include "word32.h"

void
Word32_dump(char *pattern,
            uint32_t value)
{
for(int i=0;i<32;++i)
  {
  pattern[i]=Word32_test(value,31-i) ? BIT_ONE_CHAR : BIT_ZERO_CHAR;
  }
}

uint32_t
Word32_load(const char *pattern)
{
uint32_t result=0;
for(int i=0;i<32;++i)
  {
  if(pattern[i]==BIT_ONE_CHAR)
    { Word32_set(&result,31-i); }
  }
return result;
}
```

10. L06_Maths : Opérations mathématiques

Motivation :

Il est souvent nécessaire d'avoir recours au langage C lorsqu'il est question de traiter des problèmes très calculatoires afin de réduire autant que possible la durée de ces calculs. Jusqu'alors, ces derniers reposaient sur l'usage des opérations arithmétiques de base sur des entiers et des réels, ainsi que sur des opérations logiques ou *bit-à-bit*.

Pour aller au-delà, le langage C propose en standard une bibliothèque de fonctions mathématiques usuelles, principalement dédiée aux réels. Un aspect souvent associée aux problèmes mathématiques concerne la générations de valeurs aléatoires ; la bibliothèque standard propose des fonctionnalités à cet effet.

Consignes :

Au delà de la découverte très guidée de ces quelques nouvelles fonctionnalités, le travail demandé ici nécessitera d'être autonome dans la réutilisation des notions déjà acquises (tableaux, passages de paramètres...) afin de les appliquer à un cas d'usage concret. Il faudra également savoir consulter la documentation. Veilliez à suivre scrupuleusement chacune des étapes en vous assurant à chaque fois d'avoir bien observé et compris ce qui vous est décrit.

10.1. Mise en place du programme

Ouvrez un terminal et grâce à la ligne de commande du *shell*, placez vous dans le répertoire de travail choisi pour ce sujet. Saisissez par exemple :

```
$ mkdir S3PRC_L06_Maths ↲
$ cd S3PRC_L06_Maths ↲
```

Placez dans ce répertoire le fichier `GNUmakefile` générique mis à votre disposition.

(http://www.enib.fr/~harrouet/Data/Courses/GNUmakefile)

Nous suivrons en effet la démarche de programmation modulaire présentée au chapitre 2.

Le programme principal sera rédigé dans le fichier `prog06.c` et appellera une fonction `test_maths()` qui sera réalisée dans ce même fichier. Cette dernière qui, n'attend aucun paramètre et ne renvoie aucun résultat, se contentera pour l'instant d'afficher son nom.

Fabriquez ce programme et exécutez-le afin de vérifier que tout est en place. En l'état, il ne fait qu'afficher le nom de la fonction `test_maths()` :

```
$ make ↲
$ ./prog06 ↲
```

10.2. La bibliothèque mathématique

Bien que nous ayons vu en 4.4.3 comment calculer une fonction mathématique à l'aide d'un procédé itératif, il existe cependant un moyen beaucoup plus efficace. En effet, les processeurs des machines informatiques sont généralement équipés d'un jeu d'instructions correspondant

aux fonctions mathématiques usuelles ; ils sont capables de les évaluer bien plus rapidement que nous le ferions avec un programme.

La langage C fournit en standard une bibliothèque dédiée à l'invocation de ces fonctionnalités qui sont déclarées par le fichier d'en-tête standard *math.h*.

(http://en.cppreference.com/w/c/numeric/math)

10.2.1. Mise en œuvre de la bibliothèque mathématique

Nous cherchons ici à compléter la fonction *test_maths()* afin qu'elle affiche clairement le résultat de quelques fonctions mathématiques usuelles.

Commençons par calculer et afficher le résultat de la fonction *cosinus* appliquée à la valeur π/3.

(http://en.cppreference.com/w/c/numeric/math/cos)

Sa documentation nous apprend qu'elle existe en différentes versions selon que nous souhaitons calculer avec des données de type *float*, *double* ou *long double* (voir en 4.4). Cette distinction est similaire pour toutes les fonctions mathématiques et repose sur la dernière lettre qui est ajoutée au nom de la fonction : *f* pour *float*, aucune pour double ou *l* pour *long double*. Ici nous utiliserons le type *double*, donc la fonction *cos()*.

Son usage est trivial : nous lui transmettons un réel et nous obtenons son résultat que nous nous contentons d'afficher. Toutefois, tout le monde ne connaît pas par cœur la valeur de π avec suffisamment de décimales significatives. Beaucoup d'environnements de développement font apparaître dans le fichier d'en-tête standard *math.h* une *macro* *M_PI* qui décrit cette valeur ; seulement le standard du langage C ne la mentionne pas et selon les options de compilations cette *macro* sera rendue visible ou non[129]. Une démarche prudente, consiste donc à définir cette *macro* par nos propres moyen dans notre code source seulement si elle ne l'est pas déjà :

```
#ifndef M_PI
# define M_PI 3.14159265358979323846
#endif
```

Désormais, la fonction *cos()* peut être invoquée avec comme paramètre *M_PI/3.0*.

Relancez alors la fabrication du programme exécutable ; elle doit échouer ! Le problème ne se situe pas à la compilation (si vous avez bien inclus le fichier d'en-tête standard *math.h*), mais à l'édition de liens. En effet , la bibliothèque mathématique est bien livrée en standard avec tous les outils de développement mais, contrairement à la bibliothèque standard du langage C (qui définit *printf()*, *strlen()*, *malloc()*...), elle n'est pas automatiquement ajoutée au programme lors de son édition de liens. C'est donc à nous de spécifier à l'éditeur de liens le fait qu'il doit prendre en compte cette bibliothèque supplémentaire.

Pour ce faire, il nous faut compléter le fichier *GNUmakefile* : la toute première ligne qui contient

LDFLAGS=

doit être complétée pour contenir

LDFLAGS=-lm

La variable du fichier *GNUmakefile* est transmise à la commande invoquant l'éditeur de liens. Il comprend l'option *-lm* que nous venons d'ajouter comme : il faut lier l'exécutable produit à une bibliothèque supplémentaire (préfixe *-l*) ; il s'agit de la bibliothèque mathématique (suffixe *m* ici)[130].

Une nouvelle relance de la fabrication du programme exécutable doit désormais aboutir avec succès. Exécutez-le afin de vérifier qu'il affiche bien la valeur attendue.

129 Il existe potentiellement d'autres *macros* de cette nature : *M_E*, *M_SQRT2*...

130 L'éditeur de liens recherche alors dans les emplacements standards du système une bibliothèque nommée *libm.so*, *libm.a*, *libm.dylib*... selon le système d'exploitation utilisé.

Maintenant que nous avons réussi à utiliser une fonction de la bibliothèque mathématique, expérimentez-en quelques autres pour afficher des valeurs remarquables : la *racine carrée* de deux, l'*exponentielle* de un, le *logarithme* à base dix d'un centième…

(http://en.cppreference.com/w/c/numeric/math)

Remarquez qu'il existe aussi des fonctions à plusieurs paramètres ; c'est le cas par exemple de la fonction `atan2()` qui renvoie l'*arc-tangente* du rapport de ses deux paramètres sur le cercle trigonométrique complet[131]. Invoquez également cette fonction avec des valeurs remarquables.

Fabriquez à nouveau le programme et exécutez-le afin de vérifier que les valeurs obtenues sont conformes aux attentes.

10.2.2. Détection des valeurs extrêmes

Le programme doit maintenant être complété par un nouveau module composé du fichier de définition `realArray.c` et du fichier d'en-tête `realArray.h` qui respecteront les consignes usuelles en matière de programmation modulaire (voir le chapitre 2). Ce module doit fournir une fonction `RealArray_minMax()` dont le rôle est de parcourir (en lecture seule) un tableau de valeurs réelles (désigné par un pointeur sur la première et leur nombre), pour mettre à jour deux variables du contexte appelant afin qu'elles contiennent respectivement les valeurs minimale et maximale rencontrées dans le tableau.

Nous pourrions déterminer chacune de ces valeurs extrêmes à l'aide d'un opérateur de comparaison et de l'opérateur ternaire ; toutefois, les processeurs sont généralement pourvus d'instructions spécialisées à cet effet qui sont rendues accessibles par les fonctions `fmin()` et `fmax()` de la bibliothèque mathématique.

(http://en.cppreference.com/w/c/numeric/math/fmin)

(http://en.cppreference.com/w/c/numeric/math/fmax)

Nous utiliserons donc ces fonctions lors du parcours du tableau, en prenant soin d'utiliser initialement `DBL_MAX` comme valeur minimale et `-DBL_MAX` comme valeur maximale (voir en 4.4.2). De cette façon, toute valeur rencontrée sera respectivement inférieure et supérieure à ces valeurs initiales. Il suffit alors d'appeler ses fonctions en leur transmettant l'extrême concerné et la valeur rencontrée afin d'ajuster ce même extrême. À la sortie de la boucle, les valeurs extrêmes ont été déterminées ; il ne reste plus qu'à les transmettre au contexte appelant.

Une fonction `test_realArray()` sera réalisée dans le fichier `prog05.c` et sera appelée par le programme principal. Cette nouvelle fonction de test doit commencer par afficher son nom, déclarer un tableau de quelques dizaines de valeurs réelles non initialisées et en déterminer le nombre d'éléments `count`. Il faudra alors initialiser chaque élément `i` du tableau avec la valeur 4*π*`i`/`count`+π/4. La fonction `RealArray_minMax()` sera alors utilisée afin d'afficher les valeurs extrêmes du contenu du tableau.

Fabriquez à nouveau ce programme et exécutez-le afin de vérifier que ces deux valeurs sont bien cohérentes avec le contenu tableau.

10.2.3. Arrondi d'un réel vers un entier

La bibliothèque mathématique propose plusieurs fonctions pour déterminer une valeur entière approchée d'une valeur réelle. Parmi celles-ci nous trouvons principalement :

- `floor()`, l'arrondi à l'entier par défaut : le résultat est systématiquement inférieur ou égal au paramètre,
  (http://en.cppreference.com/w/c/numeric/math/floor)
- `ceil()`, l'arrondi à l'entier par excès : le résultat est systématiquement supérieur ou égal au paramètre,
  (http://en.cppreference.com/w/c/numeric/math/ceil)
- `trunc()`, la troncature de la partie décimale : le résultat est inférieur ou égal au paramètre s'il est positif ou bien supérieur ou égal au paramètre s'il est négatif (comme

131 Contrairement à la fonction `atan()` qui ne le fait que sur un demi cercle.

dans le cas du *cast* vers un entier),
(http://en.cppreference.com/w/c/numeric/math/trunc)

- `round()`, l'arrondi à l'entier le plus proche : si la partie décimale du paramètre est supérieure ou égale à `0.5`, alors le résultat est l'entier ayant la valeur absolue supérieure, ou celui ayant la valeur absolue inférieure dans le cas contraire.
(http://en.cppreference.com/w/c/numeric/math/round)

Remarquez que bien qu'elles déterminent toutes une valeur entière, le résultat de ces fonctions est toujours de type réel ; du point de vue du langage C, si nous souhaitons en faire un véritable entier, un *cast* est encore nécessaire.

Nous nous proposons ici d'en faire usage en ajoutant au module `realArray` la fonction `RealArray_display()` suivante :

```
void
RealArray_display(const double *data,
                  int count,
                  double minValue,
                  double maxValue)
{
const double valueRange=maxValue-minValue;
const int first=8, last=70, range=last-first+1;
const int zero=first+(int)round(range*(-minValue)/valueRange);
for(int i=0;i<count;++i)
  {
  const double value=data[i];
  const int step=first+(int)round(range*(value-minValue)/valueRange);
  const char symbol=step>zero ? '>' : step<zero ? '<' : 'X';
  for(int s=0;s<=78;++s)
    {
    char c=' ';
    if(step==s) { c=symbol; }
    else if(s==zero) { c='|'; }
    else if(((step<=s)&&(s<=zero))||((zero<=s)&&(s<=step))) { c='~'; }
    fputc(c,stdout);
    }
  fputc('\n',stdout);
  }
}
```

Elle représente sous forme de barres textuelles horizontales de longueurs variables les valeurs réelles contenues dans le tableau. Il s'agit bien d'un problème de conversion d'une valeur réelle en un nombre entier de symboles textuels pour la représenter. Nous choisissons ici l'arrondi à l'entier le plus proche afin de représenter le plus fidèlement possible la valeur réelle initiale.

La fonction `RealArray_display()` sera alors utilisée dans la fonction `test_realArray()` afin de représenter le contenu du tableau.

Fabriquez à nouveau ce programme et exécutez-le afin de vérifier que vous obtenez bien une rampe croissante cohérente avec le contenu du tableau.

Pour un affichage plus explicite, il faut maintenant compléter le module `realArray` avec une fonction `RealArray_sin()` qui remplace chaque valeur réelle du tableau qu'on lui transmet par le résultat de la fonction `sin()` (*sinus*) qui lui est appliquée. Elle sera utilisée sur le tableau de la fonction `test_realArray()` et les nouvelles valeurs extrêmes du tableau seront déterminées et affichées avant d'invoquer l'affichage sous forme de barres textuelles horizontales.

(http://en.cppreference.com/w/c/numeric/math/sin)

Fabriquez à nouveau ce programme et exécutez-le afin de vérifier que les valeurs extrêmes sont bien celles de la sinusoïde et que cette dernière effectue bien deux tours de cercle trigonométrique avec un déphasage d'un huitième de tour.

10.2.4. Les valeurs réelles anormales

Contrairement aux entiers pour lesquels toutes les combinaisons de *bits* s'interprètent comme une valeur entière, il existe des combinaisons de *bits* dans la représentation des réels qui ne correspondent à aucune valeur réelle finie.

(https://en.wikipedia.org/wiki/IEEE_floating_point)

La bibliothèque mathématique fournit notamment la fonction `fpclassify()` qui permet de classifier la représentation d'un réel selon une variété de catégories (valeur normale, infinie, *not-a-number*...)[132].

(http://en.cppreference.com/w/c/numeric/math/fpclassify)

Ceci donne l'opportunité aux fonctions mathématiques de renvoyer un résultat qui signifie clairement que nous l'utilisons en dehors de son domaine de définition. Pour expérimenter ce comportement, complétons le module `realArray` par une fonction `RealArray_sqrt()` qui remplace chaque valeur réelle du tableau qu'on lui transmet par le résultat de la fonction `sqrt()` (*racine carrée*) qui lui est appliquée.

(http://en.cppreference.com/w/c/numeric/math/sqrt)

Elle sera utilisée, après le calcul des *sinus*, sur le tableau de la fonction `test_realArray()` et les nouvelles valeurs extrêmes du tableau seront déterminées et affichées avant d'invoquer l'affichage sous forme de barres textuelles horizontales.

Fabriquez à nouveau ce programme et exécutez-le afin de vérifier que les valeurs extrêmes sont bien positives ou nulles (à cause de la *racine carrée*) et que les lobes positifs de la sinusoïde on bien été déformés par cette opération. En revanche, vous devez constater que l'affichage des barres textuelles horizontales est aberrant sur les lobes négatifs de la sinusoïde : les barres partent inexorablement vers les valeurs négatives alors que la valeur minimal détectée est proche de zéro !

Ceci est dû au fait que sur ces derniers lobes nous tentons de déterminer la racine carrée d'un nombre négatif. La fonction `sqrt()` détecte cette incohérence et fournit le résultat `NAN` (*not-a-number*) signifiant que nous sortons de son domaine de définition. Il est intéressant de remarquer que les fonctions `fmin()` et `fmax()` utilisées pour déterminer les valeurs extrêmes du tableau ne considèrent jamais cette valeur anormale comme étant inférieure ou supérieure à toute autre valeur.

Pour vous en convaincre, affichez alors simplement sous forme de texte les valeurs du tableau dans la fonction `test_realArray()`. Après une nouvelle fabrication du programme, son exécution devrait faire apparaître ces valeurs anormales avec le texte `nan`.

Pour corriger l'affichage sous forme de barres textuelles horizontales, il nous faut être en mesure de déterminer si une valeur réelle est finie ou anormale. La bibliothèque mathématique nous propose à cet effet la fonction `isfinite()`.

(http://en.cppreference.com/w/c/numeric/math/isfinite)

Modifiez alors légèrement la fonction `RealArray_display()` pour initialiser la variable `step` avec la variable `zero`, et la variable `symbol` avec le caractère littéral `?` dans le cas où la valeur réelle extraite du tableau n'est pas finie.

Fabriquez à nouveau ce programme et exécutez-le afin de vérifier que l'affichage fait apparaître des points d'interrogation pour représenter les valeurs du tableau qui ne sont pas finies.

---

132 L'option de compilation `-ffast-math` provoque une optimisation très agressive qui suppose qu'aucune valeur anormale ne peut apparaître dans les calculs ; cela rend la détection d'une telle valeur impossible.

10.3. Génération pseudo-aléatoire

Il n'est pas rare d'avoir à recours à des données générées aléatoirement. C'est le cas par exemple des simulations numériques selon la méthode de *Monte-Carlo,*
(https://en.wikipedia.org/wiki/Monte_Carlo_method)
du test de composants logiciels par la méthode de *fuzzing*,
(https://en.wikipedia.org/wiki/Fuzzing)
ou encore des jeux qui rendent l'animation des personnages et des décors imprévisibles.

La bibliothèque standard propose une fonctionnalité en ce sens.

10.3.1. Obtention de valeurs pseudo-aléatoires

Le programme principal du fichier `prog05.c` doit faire appel à une fonction `test_rand()` qui sera réalisée dans ce même fichier. Cette dernière qui, n'attend aucun paramètre et ne renvoie aucun résultat, se contentera pour l'instant d'afficher son nom ainsi qu'une dizaine de valeurs entières obtenues par la fonction standard `rand()` déclarée dans le fichier d'en-tête standard `stdlib.h`.

(http://en.cppreference.com/w/c/numeric/random/rand)
Fabriquez à nouveau ce programme et exécutez-le afin de vérifier qu'apparaît bien une séquence de valeurs entières *a priori* imprévisibles ; elles semblent donc aléatoires.

Toutefois, en relançant ce programme à plusieurs reprises, nous constatons que la séquence de valeurs entières obtenue est à chaque fois identique : le procédé sous-jacent est dit *pseudo-aléatoire*. Un tel procédé repose sur un algorithme totalement déterministe qui doit être initialisé. Pour une initialisation donnée, la séquence de valeurs obtenue sera toujours identique, mais des initialisations différentes donneront des séquences totalement différentes. Ces séquences, bien que déterministes, donnent aux êtres humains qui les observent l'illusion que les valeurs apparaissent dans un ordre complètement imprévisible. De plus, l'algorithme de génération pseudo-aléatoire est conçu de façon à ce que chaque séquence désigne toutes les valeurs possibles de manière équiprobable.

Si nous souhaitons obtenir à chaque nouvelle exécution une séquence pseudo-aléatoire différente, il nous faut initialiser explicitement le générateur avec une *graine* (*seed*). La fonction standard `srand()`, déclarée elle aussi dans le fichier d'en-tête standard `stdlib.h`, sert ce propos.

(http://en.cppreference.com/w/c/numeric/random/srand)
Seulement, le choix de cette graine est tout aussi problématique puisqu'une valeur constante spécifiée dans le code source produirait elle aussi toujours la même séquence à chaque exécution.

Une solution très commune consiste à rendre la graine dépendante du temps. Pour ceci, le fichier d'en-tête standard `time.h` déclare la fonction standard `time()` qui fournit une valeur qui évolue à chaque seconde.

(http://en.cppreference.com/w/c/chrono/time)
Il nous suffit donc d'ajouter au début de la fonction `main()` l'appel suivant :

```
srand((unsigned int)time(NULL));
```

pour que tous les tirages aléatoires de notre programme soient différents à chacune de ses exécutions[133].

Fabriquez à nouveau ce programme et exécutez-le plusieurs fois afin de vérifier que la séquence de valeurs entières pseudo-aléatoires est bien différente à chaque relance.

133 Dans un délai d'une seconde cependant ; une horloge avec une résolution plus fine pourrait tout aussi bien être utilisée si cela s'avérait nécessaire.

10.3.2. Choix de la plage de valeurs pseudo-aléatoires

La documentation de la fonction `rand()` nous indique que l'entier renvoyé est compris entre `0` et la constante `RAND_MAX` (incluse)[134] ; une telle valeur est difficilement exploitable directement. Il est en effet assez courant d'avoir à choisir une valeur dans une plage déterminée par les conditions applicatives.

Un moyen très simple d'y parvenir consiste à appliquer l'opérateur `%` (*modulo*, le reste de la division entière) au résultat de `rand()` avec comme second opérande la largeur de la plage de valeur choisie. Ainsi, l'expression `rand()%N` fournit bien un entier pseudo-aléatoire compris entre `0` et `N-1` (inclus).

Complétez alors la fonction `test_rand()` pour tirer et afficher une centaine de valeurs pouvant valoir de `0` jusqu'à `9`. Fabriquez à nouveau ce programme et exécutez-le afin de vérifier que toutes ces valeurs sont bien tirées et qu'aucune autre ne l'est.

Puisque le générateur se veut équiprobable, nous pouvons l'exploiter pour déclencher un événement avec une probabilité de la forme : `k` fois sur `N`. Il suffit pour cela d'effectuer un tirage entre `0` et `N-1` et de tester si la valeur obtenue est strictement inférieure à `k`.

Complétez alors la fonction `test_rand()` pour effectuer un million de tirages et compter ceux qui obéissent à la probabilité : trois fois sur dix. Fabriquez à nouveau ce programme et exécutez-le afin de vérifier que le nombre de tirages respectant ce critère avoisine bien les trois dixièmes du nombre total de tirages (trois-cents mille).

10.3.3. Distributions de valeurs pseudo-aléatoires réelles

Il est très courant d'avoir besoin de valeurs pseudo-aléatoires qui soient réelles. Il est même parfois nécessaire de choisir une loi de distribution de ces valeurs. Il en existe de très nombreuse ; nous n'en aborderons ici que deux très classiques.

(https://en.wikipedia.org/wiki/List_of_probability_distributions)

Le programme doit maintenant être complété par un nouveau module composé du fichier de définition `random.c` et du fichier d'en-tête `random.h` qui respecteront les consignes usuelles en matière de programmation modulaire (voir le chapitre 2). Ce module doit fournir une fonction `Random_uniform()` dont le rôle est de fournir une valeur réelle pseudo-aléatoire comprise entre `0.0` (inclus) et `1.0` (exclus). La distribution de cette fonction doit être uniforme c'est à dire que toutes les valeurs de l'intervalle sont équiprobables.

Un tel résultat est facile à obtenir puisque nous savons que la distribution de la fonction standard `rand()` est elle-même équiprobable sur l'intervalle `[0;RAND_MAX]`. Il suffit donc de diviser la valeur entière obtenue par le réel `RAND_MAX+1.0`.

Le programme principal du fichier `prog05.c` doit faire appel à une nouvelle fonction `test_distribution()` qui sera réalisée dans ce même fichier. Cette dernière qui, n'attend aucun paramètre et ne renvoie aucun résultat, commencera par afficher son nom, déclarera un tableau de quelques dizaines de réels, tous nuls initialement, et évaluera dans une variable `count` le nombre d'éléments de ce tableau. Ce tableau servira à déterminer l'histogramme d'un million de tirages aléatoires avec la fonction `Random_uniform()`. Pour cela, le résultat du tirage (dans `[0.0;1.0[`) est multiplié par `count` et la partie entière de cette valeur désigne une case du tableau (dans `[0;count-1]`) à laquelle on ajoute la valeur un millionième. La fonction `RealArray_display()` servira alors à visualiser le tableau avec `0.0` comme valeur minimale et `1.0/count` comme valeur maximale.

Fabriquez à nouveau ce programme et exécutez-le afin de vérifier que la distribution des valeurs aléatoires tirées est bien uniformément répartie dans toutes les cases du tableau.

Une seconde distribution aléatoire sera réalisée dans le module `random` par la fonction `Random_gaussian()`. Il s'agit d'une gaussienne de moyenne `0.0` et d'écart-type `1.0`.

---

134 La constante `RAND_MAX` est une *macro* du fichier d'en-tête standard `stdlib.h`. Sa valeur est dépendante de la plateforme d'exécution.

(https://en.wikipedia.org/wiki/Normal_distribution)

Elle génère une telle valeur aléatoire à partir de deux valeurs aléatoire uniformes selon la méthode de *Box-Muller*.

(https://en.wikipedia.org/wiki/Box-Muller_transform)

```
double // random value with mean 0.0 and standard-deviation 1.0
Random_gaussian(void)
{
const double r1=Random_uniform();
const double r2=Random_uniform();
return sin(2.0*M_PI*r1)*sqrt(-2.0*log(1.0-r2)); // Box-Muller method
}
```

La fonction `test_distribution()` doit être à nouveau complétée en réinitialisant chaque élément du tableau à `0.0` et en constituant à nouveau un histogramme d'un million de tirages provenant de la fonction `Random_gaussian()`. Toutefois, contrairement à la distribution uniforme qui borne l'intervalle de ses valeurs possibles, la distribution gaussienne délivre des valeurs aléatoires non bornées. Pour que cette gaussienne recouvre correctement la largeur de l'histogramme, nous choisissons une moyenne `mean=0.5*count` et un écart-type `stdDev=0.15*count` ; la valeur aléatoire retenue sera alors `mean+stdDev*Random_gaussian()`. Puisque celle-ci n'est pas bornée, il faudra veiller à n'ajouter un millionième à l'histogramme que si la partie entière de cette valeur est bien dans l'intervalle `[0;count-1]`. La fonction `RealArray_display()` servira à nouveau à visualiser le tableau avec `0.0` comme valeur minimale et `1.0/sqrt(2.0*M_PI*stdDev*stdDev)` (le sommet de la gaussienne) comme valeur maximale.

Fabriquez à nouveau ce programme et exécutez-le afin de vérifier que la distribution des valeurs aléatoires tirées a bien la répartition gaussienne attendue.

10.4. Résumé

Ici s'achève cette sixième séance pratique dans laquelle nous avons utilisé les fonctions mathématiques concernant les réels et la génération de valeurs pseudo-aléatoires.

La mise en œuvre des fonctionnalités mathématique repose sur l'inclusion du fichier d'en-tête standard `math.h` et nécessite l'édition de lien explicite avec la bibliothèque mathématique standard. Parmi leur très grande variété, certaines concernent l'évaluation de fonctions mathématiques usuelles, d'autres sont dédiées à l'évaluation des extrema, aux arrondis ou encore à la détection des valeurs non finies.

Les fonctionnalités élémentaires dédiées aux tirages pseudo-aléatoires font partie intégrante de la bibliothèque standard du langage C et sont déclarées dans le fichier d'en-tête standard `stdlib.h`. Le choix d'une graine initiale détermine complètement la séquence de valeurs entières pseudo-aléatoires qui sera générée. Différentes distributions de probabilités peuvent être obtenues par des transformations mathématiques appliquées à cette distribution uniforme.

Voici le contenu des fichiers réalisés dans ce sujet, tels qu'ils doivent être si vous avez suivi scrupuleusement les recommandations qui vous ont été données à chaque étape.

```
//---- prog06.c ----
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "realArray.h"
#include "random.h"

#ifndef M_PI
# define M_PI 3.14159265358979323846
#endif

void
test_maths(void)
{
printf("\n~~~~ %s() ~~~~\n",__func__);
printf("cos(M_PI/3.0)=%g\n",cos(M_PI/3.0));
printf("sqrt(2.0)=%g\n",sqrt(2.0));
printf("exp(1.0)=%g\n",exp(1.0));
printf("log10(0.01)=%g\n",log10(0.01));
printf("atan2(-1.0,1.0)=%g\n",atan2(-1.0,1.0));
}

void
test_realArray(void)
{
printf("\n~~~~ %s() ~~~~\n",__func__);
double values[80];
int count=(int)(sizeof(values)/sizeof(values[0]));
for(int i=0;i<count;++i)
  { values[i]=4.0*M_PI*i/count+M_PI/4.0; }
double minValue,maxValue;
RealArray_minMax(values,count,&minValue,&maxValue);
printf("min=%g max=%g\n",minValue,maxValue);
RealArray_display(values,count,minValue,maxValue);
printf("sin():\n");
RealArray_sin(values,count);
RealArray_minMax(values,count,&minValue,&maxValue);
printf("min=%g max=%g\n",minValue,maxValue);
RealArray_display(values,count,minValue,maxValue);
printf("sqrt(sin()):\n");
RealArray_sqrt(values,count);
RealArray_minMax(values,count,&minValue,&maxValue);
printf("min=%g max=%g\n",minValue,maxValue);
RealArray_display(values,count,minValue,maxValue);
for(int i=0;i<count;++i)
  { printf("%g ",values[i]); }
printf("\n");
}



                                                                    // ...(1/2)...
```

```
void                                                                    // ...(2/2)...
test_rand(void)
{
printf("\n~~~~ %s() ~~~~\n",__func__);
for(int i=0;i<10;++i)
  { printf("%d\n",rand()); }
for(int i=0;i<100;++i)
  { printf("%d ",rand()%10); }
printf("\n");
int zeroCount=0;
const int drawCount=1000000;
for(int i=0;i<drawCount;++i)
  { if((rand()%10)<3) { ++zeroCount; } }
printf("%d zero out of %d draws\n",zeroCount,drawCount);
}

void
test_distribution(void)
{
printf("\n~~~~ %s() ~~~~\n",__func__);
const int drawCount=1000000;
double values[40]={ 0.0 };
int count=(int)(sizeof(values)/sizeof(values[0]));
printf("uniform distribution:\n");
for(int i=0;i<drawCount;++i)
  {
  const int r=(int)(count*Random_uniform());
  values[r]+=1.0/drawCount;
  }
RealArray_display(values,count,0.0,1.0/count);
printf("gaussian distribution:\n");
for(int i=0;i<count;++i)
  { values[i]=0.0; }
const double mean=0.5*count;
const double stdDev=0.15*count;
for(int i=0;i<drawCount;++i)
  {
  const int r=(int)(mean+stdDev*Random_gaussian());
  if((r>=0)&&(r<count))
    { values[r]+=1.0/drawCount; }
  }
RealArray_display(values,count,0.0,1.0/sqrt(2.0*M_PI*stdDev*stdDev));
}

int
main(void)
{
srand((unsigned int)time(NULL));
test_maths();
test_realArray();
test_rand();
test_distribution();
return 0;
}
```

```
//---- realArray.h ----
#ifndef REALARRAY_H
#define REALARRAY_H 1

void
RealArray_minMax(const double *data,
                 int count,
                 double *out_minValue,
                 double *out_maxValue);

void
RealArray_display(const double *data,
                  int count,
                  double minValue,
                  double maxValue);

void
RealArray_sin(double *data,
              int count);

void
RealArray_sqrt(double *data,
               int count);

#endif // REALARRAY_H
```

```
//---- realArray.c ----
#include "realArray.h"
#include <float.h>
#include <stdio.h>
#include <math.h>

void
RealArray_minMax(const double *data,
                 int count,
                 double *out_minValue,
                 double *out_maxValue)
{
double minValue=DBL_MAX, maxValue=-DBL_MAX;
for(int i=0;i<count;++i)
  {
  const double value=data[i];
  minValue=fmin(minValue,value);
  maxValue=fmax(maxValue,value);
  }
//-- store out-parameters --
*out_minValue=minValue;
*out_maxValue=maxValue;
}

void
RealArray_display(const double *data,
                  int count,
                  double minValue,
                  double maxValue)                                    // ...(1/2)...
```

```
{                                                                                // ...(2/2)...
const double valueRange=maxValue-minValue;
const int first=8, last=70, range=last-first+1;
const int zero=first+(int)round(range*(-minValue)/valueRange);
for(int i=0;i<count;++i)
  {
  const double value=data[i];
  int step=zero;
  char symbol='?';
  if(isfinite(value))
    {
    step=first+(int)round(range*(value-minValue)/valueRange);
    symbol=step>zero ? '>' : step<zero ? '<' : 'X';
    }
  for(int s=0;s<=78;++s)
    {
    char c=' ';
    if(step==s) { c=symbol; }
    else if(s==zero) { c='|'; }
    else if(((step<=s)&&(s<=zero))||((zero<=s)&&(s<=step))) { c='~'; }
    fputc(c,stdout);
    }
  fputc('\n',stdout);
  }
}

void
RealArray_sin(double *data,
              int count)
{
for(int i=0;i<count;++i)
  { data[i]=sin(data[i]); }
}

void
RealArray_sqrt(double *data,
               int count)
{
for(int i=0;i<count;++i)
  { data[i]=sqrt(data[i]); }
}
```

```
//---- random.h ----
#ifndef RANDOM_H
#define RANDOM_H 1

double // random value in [0.0,1.0)
Random_uniform(void);

double // random value with mean 0.0 and standard-deviation 1.0
Random_gaussian(void);

#endif // RANDOM_H
```

```
//---- random.c ----
#include "random.h"
#include <stdlib.h>
#include <math.h>

#ifndef M_PI
# define M_PI 3.14159265358979323846
#endif

double // random value in [0.0,1.0)
Random_uniform(void)
{
return rand()/(RAND_MAX+1.0);
}

double // random value with mean 0.0 and standard-deviation 1.0
Random_gaussian(void)
{
const double r1=Random_uniform();
const double r2=Random_uniform();
return sin(2.0*M_PI*r1)*sqrt(-2.0*log(1.0-r2)); // Box-Muller method
}
```

11. L07_FnctPtr : Pointeurs sur fonctions

Motivation :

Toutes les fonctions réalisées jusqu'alors produisaient des résultats qui ne dépendaient que des données qui leur étaient transmises. Les fonctions et la façon dont elles s'enchaînent dans le déroulement d'un programme semblent dont intrinsèquement figées, contrairement aux données qui peuvent être produites dynamiquement.

Toutefois, comme vous le découvrirez dans les matières S5-MIP et S6-MIP de l'ENIB, les fonctions sont des séquences d'octets, représentant les instructions destinées au processeur, qui sont placées dans la mémoire de la machine informatique ; elles ont donc une adresse et peuvent par conséquent être désignées par cette dernière. Les types qui sont des pointeurs sur fonctions permettent à ce propos de créer des variables qui désignent tantôt une fonction tantôt une autre et donc de provoquer l'appel de l'une ou de l'autre comme s'il s'agissait de simples données. Ceci peut être exploité pour réaliser des algorithmes génériques qui invoquent des traitements spécifiquement choisis selon les besoins applicatifs.

Consignes :

Au delà de la découverte très guidée de ces quelques nouvelles fonctionnalités, le travail demandé ici nécessitera d'être autonome dans la réutilisation des notions déjà acquises afin de les appliquer à un cas d'usage concret. Il faudra également savoir consulter la documentation. Veilliez à suivre scrupuleusement chacune des étapes en vous assurant à chaque fois d'avoir bien observé et compris ce qui vous est décrit.

11.1. Mise en place du programme

Ouvrez un terminal et grâce à la ligne de commande du *shell*, placez vous dans le répertoire de travail choisi pour ce sujet. Saisissez par exemple :

```
$ mkdir S3PRC_L07_FnctPtr ↲
$ cd S3PRC_L07_FnctPtr ↲
```

Placez dans ce répertoire le fichier `GNUmakefile` générique mis à votre disposition.

(http://www.enib.fr/~harrouet/Data/Courses/GNUmakefile)

Nous suivrons en effet la démarche de programmation modulaire présentée au chapitre 2.

Nous utilisons comme prétexte l'intégration de fonctions mathématiques. La fonction `Integrate_sqrt()` reproduite juste après utilise la méthode des rectangles pour approximer l'intégrale de la fonction *racine carrée*.

(https://en.wikipedia.org/wiki/Rectangle_method)

Cette fonction `Integrate_sqrt()` sera déclarée dans le fichier d'en-tête `integrate.h` et définie dans le fichier `integrate.c` (par simple copier/coller du code suivant).

```
double
Integrate_sqrt(double xMin,
               double xMax,
               int stepCount)
{
const double dx=(xMax-xMin)/stepCount;
double sum=0.0, x=xMin+0.5*dx;
for(int i=0;i<stepCount;++i, x+=dx)
  {
  const double value=sqrt(x);
  if(isfinite(value)) { sum+=value; }
  }
return dx*sum;
}
```

Le programme principal sera rédigé dans le fichier *prog07.c* et appellera la fonction *test_Integrate_math()* qui sera réalisée dans ce même fichier et qui, après avoir affiché son nom, utilisera la fonction *Integrate_sqrt()* sur l'intervalle *[0.0;1.0]* avec un millier de pas et en affichera le résultat. Remarquez que, comme vu en 10.2.1, l'usage de la fonction mathématique *sqrt()* nécessite l'édition de liens avec la bibliothèque mathématique : la variable *LDFLAGS* du fichier *GNUmakefile* devra à nouveau être complétée avec l'option *-lm*.

Fabriquez ce programme et exécutez-le afin de vérifier que tout est en place :

```
$ make ↲
$ ./prog07 ↲
```

En l'état, il ne fait qu'afficher le nom de la fonction *test_Integrate_math()* et le résultat de l'évaluation de l'intégrale choisie. Vérifiez que ce résultat est correct, sachant qu'une primitive de la *racine carrée* est $\frac{x^{3/2}}{3/2}$.

Complétez alors le module *integrate* pour fournir une nouvelle fonction *Integrate_log()* qui s'inspire de la fonction *Integrate_sqrt()* pour évaluer l'intégrale de la fonction *log()* selon le même procédé. Il faudra ajouter à la fonction *test_Integrate_math()* le test de cette nouvelle fonction sur le même intervalle. Après une nouvelle fabrication du programme exécutable et sa relance, vérifiez que ce nouveau résultat est correct, sachant qu'une primitive du *logarithme* est $x.log\,x - x$.

11.2. Déclaration et utilisation d'un pointeur sur fonctions

Une comparaison attentive des deux fonctions d'intégration précédemment réalisées nous montre une identité presque parfaite. En effet, l'algorithme est strictement le même dans les deux cas : seule la fonction mathématique invoquée diffère. Il serait dont souhaitable d'avoir un unique algorithme recevant en paramètre la fonction mathématique à intégrer.

Le langage C propose justement des types permettant de désigner des fonctions à l'aide de variables : les pointeurs sur fonctions. Seulement, toutes les fonctions ne sont pas interchangeables : seules le sont celles qui ont des prototypes identiques, c'est la raison pour laquelle un type de pointeur sur fonctions spécifie très précisément le prototype des fonctions qu'il peut désigner.

(http://en.cppreference.com/w/c/language/pointer#Pointers_to_functions)

Dans le cas de notre exemple, nous traitons de fonctions mathématiques simples. De telles fonctions attendent un paramètre réel et fournissent un résultat réel ; elles ont donc un prototype semblable à :

`double nom_de_la_fonction(double)`

Pour déclarer une variable de type pointeur sur de telles fonctions, la notation est la suivante :

`double (*nom_de_la_variable)(double);`

Contrairement aux déclarations de variables de types simples (comme `int` ou `double`), le nom de la variable ne suit pas le nom du type mais apparaît en son milieu. Par rapport au prototype des fonctions, nous constatons que nous substituons simplement le `nom_de_la_fonction` par la notation `(*nom_de_la_variable)` (les parenthèses et l'étoile doivent obligatoirement apparaître comme indiqué). De part et d'autre nous retrouvons les paramètres et le type de retour comme dans le cas du prototype d'une fonction.

Comme tout autre pointeur (voir en 3.2.1), une telle variable de type pointeur sur fonctions doit être initialisée pour désigner une information utile (une fonction ici) ou bien pour signifier très clairement qu'elle ne désigne rien (pointeur nul).

```
double (*math1)(double)=NULL; // la variable math1 ne désigne aucune fonction mathématique
double (*math2)(double)=sqrt; // la variable math2 désigne la fonction mathématique sqrt()
double (*math3)(double)=log;  // la variable math3 désigne la fonction mathématique log()
int (*cmp1)(const char *,const char *)=NULL;   // la variable cmp1 ne désigne aucune fonction
int (*cmp2)(const char *,const char *)=strcmp; // la variable cmp2 désigne la fonction strcmp()
```

Sur ces exemples d'initialisation, nous constatons que le nom d'une fonction (sans les parenthèses qui en matérialisent l'appel, afin de transmettre les éventuels paramètres) correspond à l'adresse de cette fonction[135] ; c'est cette information qui est mémorisée dans un pointeur sur fonctions.

Un pointeur sur fonctions est un pointeur comme un autre : il peut être affecté, passé en paramètre à une fonction, renvoyé comme résultat d'une fonction, comparé à un autre pour déterminer s'il est identique... seules l'indexation et l'arithmétique ne s'y appliquent pas. Bien entendu lorsqu'un pointeur sur fonctions est recopié, il doit l'être vers un pointeur sur fonctions correspondant à la même signature (de la même façon qu'un pointeur sur entiers ne peut pas être implicitement converti vers un pointeur sur réels). Il est éventuellement possible d'utiliser un *cast* vers un pointeur sur fonctions correspondant à un autre prototype mais il est extrêmement probable que cela conduise à un comportement indéfini du programme.

L'intérêt essentiel de mémoriser l'adresse d'une fonction dans un tel pointeur est de pouvoir invoquer ultérieurement cette même fonction en utilisant le pointeur qui aura été transmis à une partie du programme qui en a besoin.

```
if(math3)         // si une fonction est désignée par ce pointeur, nous pouvons l'invoquer
  { y=math3(x); } // en lui transmettant ses paramètres et obtenir son résultat
if(cmp2)          // même chose avec un autre type de pointeur sur fonctions
  { r=cmp2(str1,str2); }
```

Nous constatons qu'invoquer une fonction via un pointeur qui la désigne utilise exactement la même notation qu'un appel de fonction habituel[136].

11.3. Généralisation d'un traitement

Revenons maintenant à notre problème d'intégration de fonctions mathématiques pour ajouter au module `integrate` une fonction `Integrate_math()` très semblable aux deux précédentes (qui seront conservées uniquement pour mémoire désormais). Elle attendra un paramètre supplémentaire pour désigner la fonction mathématique à appeler. Son algorithme fera simplement usage de ce pointeur sur fonctions en lieu et place des fonctions `sqrt()` et `log()` utilisées jusqu'alors.

---

135 L'opérateur de référencement `&` peut également être utilisé devant le nom de la fonction dont on prend l'adresse (`&sqrt`, `&log`...) mais cette notation n'apporte rien dans le cas présent.

136 Une écriture alternative répète explicitement les parenthèses et l'étoile utilisées lors de la déclaration de la variable (`y=(*math3)(x);`, `r=(*cmp2)(str1,str2);`...) mais cette notation n'apporte rien dans le cas présent.

Il faudra ajouter à la fonction *test_Integrate_math()* du programme principal deux appels à cette nouvelle fonction sur le même intervalle que pour les tests précédents en leur transmettant respectivement l'adresse des fonctions *sqrt()* et *log()*. Après une nouvelle fabrication du programme exécutable et sa relance, vérifiez que les résultats produits par ces deux invocations du traitement générique sont bien identiques à ceux précédemment obtenus avec les traitements spécifiques.

Nous avons utilisé ici des fonctions mathématiques déjà fournies en standard mais rien ne nous empêche d'en utiliser de toutes autres du moment qu'elles respectent le prototype choisi pour le pointeur sur fonctions retenu.

Complétez alors le programme principal avec une fonction *unitGaussian()* qui, elle aussi, attend un paramètre réel et renvoie un résultat réel. Elle utilisera la formule suivante :

$$f(x \mid \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \, e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Il s'agit d'une distribution gaussienne dans laquelle *μ* et *σ* sont des paramètres représentant respectivement la moyenne et l'écart-type.

(https://en.wikipedia.org/wiki/Normal_distribution)

Dans *unitGaussian()* nous les fixerons à *0.0* et *1.0* respectivement afin que l'unique paramètre de la fonction corresponde à la variable *x* de la formule.

Il faut alors ajouter à la fonction *test_Integrate_math()* du programme principal un nouvel appel à la fonction *Integrate_math()* pour qu'elle intègre *unitGaussian()* sur l'intervalle *[0.0;1.0]*. Après une nouvelle fabrication du programme exécutable et sa relance, vous devriez être en mesure de vérifier que le résultat obtenu correspond bien aux 34.1 % attendus pour un unique écart-type après la moyenne sur une gaussienne.

(https://en.wikipedia.org/wiki/Standard_deviation)

11.4. Paramétrage des traitements génériques

Nous disposons dorénavant, avec la fonction *Integrate_math()*, d'un moyen générique d'évaluer l'intégrale d'une fonction mathématique quelconque. Cependant, le seul paramètre attendu par la fonction mathématique est la variable selon laquelle l'intégration a lieu. Si d'autres paramètres doivent être pris en compte dans cette fonction mathématique (moyenne et écart-type d'une gaussienne, amplitude, fréquence et phase d'une sinusoïde...) il est nécessaire de les fournir comme des constantes littérales dans le code de ces fonctions (c'est ce que nous avons fait dans *unitGaussian()*). Néanmoins, si ces paramètres supplémentaires ne sont pas connus à la compilation mais découverts à l'exécution, leur expression littérale préalable sera rendue impossible.

Il nous faut donc étendre le prototype des fonctions mathématiques que nous souhaitons intégrer pour transmettre les paramètres requis. Puisque ces fonctions seront appelées par la fonction générique d'intégration, cette dernière devra également en avoir connaissance afin de les transmettre à la fonction à intégrer. Seulement, la fonction générique d'intégration ne peut connaître le nombre et le type de ces paramètres puisqu'ils peuvent être tout à fait différents d'une fonction mathématique à une autre. Cette dernière remarque nous contraint à fournir à la fonction générique un moyen de transmettre *aveuglément* (sans en connaître les détails) les paramètres que la fonction appelée requiert.

Le type *void ** (déjà rencontré en 6.3.1) représente un pointeur vers une donnée de type quelconque. Le seul usage que l'on puisse en faire directement est le transfert ou la mémorisation d'une adresse. Néanmoins, une conversion explicite d'une telle adresse vers un pointeur sur un type connu permet à nouveau de faire usage de la donnée désignée. C'est donc ce procédé qui servira à confier à la fonction générique l'adresse d'une donnée qu'elle transmettra *aveuglément* à la fonction qu'elle doit invoquer par le pointeur sur fonctions. Les fonctions alors invoquées doivent donc désormais accepter également un paramètre

supplémentaire de type `void *` et doivent savoir le convertir en un pointeur sur le type effectif de la donnée désignée afin d'en faire l'usage attendu (les paramètres supplémentaires de nos fonctions mathématiques ici).

Les fonctions mathématiques que nous souhaitons intégrer auront désormais le prototype suivant :

```
double nom_de_la_fonction(double, void *)
```

Un pointeur sur fonctions correspondant devrait donc être déclaré comme ceci :

```
double (*nom_de_la_variable)(double, void *);
```

Remarquez que les types des pointeurs sur fonctions sont de manière générale difficilement lisibles et que la situation s'aggrave lorsque le prototype se complexifie. Il est alors d'usage très courant d'utiliser le mot clef `typedef` (voir en 1.3.2) pour leur attribuer une fois pour toutes un nom lisible[137].

```
typedef double (*nom_du_type)(double, void *);
```

Le `nom_du_type` inventé sera alors utilisé aussi simplement que le nom d'un type de base partout où il sera nécessaire (paramètre d'une fonction générique, membre d'une structure...).

```
nom_du_type variable;
void une_fonction(nom_du_type fnct, void *params);
```

La donnée désignée par le paramètre de type `void *` peut être quelconque : il peut s'agir d'un type de base ou d'un type élaboré, comme une structure, afin de transmettre plusieurs informations à la fois. Pour revenir à notre cas applicatif, nous souhaitons maintenant paramétrer notre fonction gaussienne selon une moyenne et un écart-type. Pour ce faire, nous définissons dans le programme principal un type de structure contenant ces deux paramètres et une fonction `gaussian()` semblable à `unitGaussian()` mais attendant un paramètre supplémentaire de type `void *`. La première opération de cette fonction consistera à considérer son paramètre de type `void *` comme un pointeur sur la structure contenant les paramètres afin d'en récupérer les valeurs. Elle reprendra alors les formules précédentes (voir en 11.3) en considérant ces paramètres.

Le module `integrate` sera alors complété par la définition avec `typedef` d'un type `ParamMathFnct` correspondant à un pointeur sur fonctions ayant le nouveau prototype attendu et par une nouvelle fonction `Integrate_param()` semblable à `Integrate_math()` mais faisant usage de ce nouveau type de pointeur sur fonctions. Il lui faudra donc un paramètre `void *` supplémentaire qui sera transmis à chaque invocation de la fonction pointée.

Une fonction `test_Integrate_param()` devra être ajoutée au programme principal. Elle attend deux paramètres réels représentant une moyenne et un écart-type qu'elle affichera en plus de son nom. Ces deux paramètres permettront d'initialiser une structure servant à contenir les paramètres supplémentaires attendus par la fonction `gaussian()`. Il suffira alors d'invoquer la fonction `Integrate_param()` pour évaluer l'intégrale sur l'intervalle `[0.0;1.0]` de la fonction `gaussian()` en prenant en compte les paramètres fournis par la structure.

La fonction principale doit réaliser un premier appel à la fonction `test_Integrate_param()` avec les valeurs `0.0` et `1.0` comme moyenne et écart-type, puis un second appel avec `1.0` et `0.5` comme nouvelles valeurs. Après une nouvelle fabrication du programme exécutable et sa relance, vous devriez être en mesure de vérifier que le résultat obtenu pour le premier de ces appels est identique à ce que nous obtenions avec `unitGaussian()` et que le résultat du second appel correspond bien aux 47.7 % attendus pour deux écarts-types avant la moyenne sur une gaussienne.

(https://en.wikipedia.org/wiki/Standard_deviation)

137 Essayez d'imaginer à quoi ressembleraient, sans l'utilisation de `typedef`, les déclarations d'un tableau de pointeurs sur fonctions, d'une fonction renvoyant un pointeur sur fonctions, d'une fonction renvoyant un tableau de pointeurs sur fonctions, d'un pointeur sur fonctions renvoyant un pointeur sur fonctions... L'usage de `typedef` améliore considérablement la lisibilité du code ; il faut l'utiliser systématiquement !

Nous disposons dorénavant, à travers les fonctions `Integrate_math()` et `Integrate_param()` de deux fonctions d'intégration génériques. La première s'applique à des fonctions simples dont le seul paramètre est la variable d'intégration, la seconde permet de prendre en compte des paramètres supplémentaires quelconques liés au contexte applicatif.

11.5. Résumé

Ici s'achève cette septième séance pratique dans laquelle nous avons découvert les pointeurs sur fonctions. Nous nous sommes appuyés dans le cas présent sur un unique exemple applicatif particulier, l'intégration de fonctions mathématiques, mais cette notion est utilisable dans bien d'autres contextes.

À chaque fois que nous savons exprimer un algorithme générique qui s'appuie sur l'invocation d'une fonctionnalité spécifique, cette dernière peut être transmise à l'algorithme générique par un pointeur sur fonctions, éventuellement accompagné d'un pointeur de type `void *` pour lui fournir des paramètres applicatifs supplémentaires sans que l'algorithme générique n'ait à en connaître les détails. Voici quelques autres cas applicatifs très classiques :

- appliquer un traitement à tous les éléments d'un ensemble (tous les éléments d'un tableau, tous les nœuds d'un arbre...) : le parcours est générique, le traitement est spécifique,
- trier un ensemble d'éléments (un tableau par exemple) : l'algorithme de tri est générique, le critère de comparaison est spécifique,
  (http://en.cppreference.com/w/c/algorithm/qsort)
- réagir à un événement en programmation événementielle (réception d'une donnée par un moyen de communication, sollicitation d'une interface graphique au clavier ou à la souris...) : la détection et la délivrance de l'événement sont génériques, la réaction à adopter lorsqu'il survient est spécifique,
- ...

En langage C++, même si les pointeurs sur fonctions restent utilisables, il existe encore d'autres facilités (*templates*, *lambda-closures...*) qui encouragent fortement l'écriture de fonctionnalités génériques réutilisables dans des contextes variés.

Voici le contenu des fichiers réalisés dans ce sujet, tels qu'ils doivent être si vous avez suivi scrupuleusement les recommandations qui vous ont été données à chaque étape.

```
//---- prog07.c ----
#include <stdio.h>
#include <math.h>
#include "integrate.h"

#ifndef M_PI
# define M_PI 3.14159265358979323846
#endif

double
unitGaussian(double x)
{
const double mean=0.0, standardDeviation=1.0;
x-=mean;
const double factor=(1.0/sqrt(2.0*M_PI))/standardDeviation;
const double exponent=-(x*x)/(2.0*standardDeviation*standardDeviation);
return factor*exp(exponent);
}

void
test_Integrate_math(void)
{
printf("\n~~~~ %s() ~~~~\n",__func__);
const double xMin=0.0, xMax=1.0;
const int stepCount=1000;
printf("Integrate_sqrt() in [%g;%g] --> %g\n",
       xMin,xMax,Integrate_sqrt(xMin,xMax,stepCount));
printf("Integrate_log() in [%g;%g] --> %g\n",
       xMin,xMax,Integrate_log(xMin,xMax,stepCount));
printf("Integrate_math(sqrt) in [%g;%g] --> %g\n",
       xMin,xMax,Integrate_math(xMin,xMax,stepCount,sqrt));
printf("Integrate_math(log) in [%g;%g] --> %g\n",
       xMin,xMax,Integrate_math(xMin,xMax,stepCount,log));
printf("Integrate_math(unitGaussian) in [%g;%g] --> %g\n",
       xMin,xMax,Integrate_math(xMin,xMax,stepCount,unitGaussian));
}

typedef struct
{
double mean,standardDeviation;
} GaussianParameters;

double
gaussian(double x,
         void *mathParams)
{
const GaussianParameters gp=*(const GaussianParameters *)mathParams;
x-=gp.mean;
const double factor=(1.0/sqrt(2.0*M_PI))/gp.standardDeviation;
const double exponent=-(x*x)/(2.0*gp.standardDeviation*gp.standardDeviation);
return factor*exp(exponent);
}
                                                                    // ...(1/2)...
```

```
void                                                                    // ...(2/2)...
test_Integrate_param(double mean,
                     double standardDeviation)
{
printf("\n~~~~ %s(%g,%g) ~~~~\n",__func__,mean,standardDeviation);
const double xMin=0.0, xMax=1.0;
const int stepCount=1000;
GaussianParameters gp={ mean, standardDeviation };
printf("Integrate_param(gaussian) in [%g;%g] --> %g\n",
       xMin,xMax,Integrate_param(xMin,xMax,stepCount,gaussian,&gp));
}

int
main(void)
{
test_Integrate_math();
test_Integrate_param(0.0,1.0);
test_Integrate_param(1.0,0.5);
return 0;
}
```

```
//---- integrate.h ----
#ifndef INTEGRATE_H
#define INTEGRATE_H 1

typedef double (*ParamMathFnct)(double, void *);

double
Integrate_sqrt(double xMin,
               double xMax,
               int stepCount);

double
Integrate_log(double xMin,
              double xMax,
              int stepCount);

double
Integrate_math(double xMin,
               double xMax,
               int stepCount,
               double (*fnct)(double));

double
Integrate_param(double xMin,
                double xMax,
                int nbSteps,
                ParamMathFnct fnct,
                void *mathParams);

#endif // INTEGRATE_H
```

```
//---- integrate.c ----
#include "integrate.h"
#include <math.h>

double
Integrate_sqrt(double xMin,
               double xMax,
               int stepCount)
{
const double dx=(xMax-xMin)/stepCount;
double sum=0.0, x=xMin+0.5*dx;
for(int i=0;i<stepCount;++i, x+=dx)
  {
  const double value=sqrt(x);
  if(isfinite(value)) { sum+=value; }
  }
return dx*sum;
}

double
Integrate_log(double xMin,
              double xMax,
              int stepCount)
{
const double dx=(xMax-xMin)/stepCount;
double sum=0.0, x=xMin+0.5*dx;
for(int i=0;i<stepCount;++i, x+=dx)
  {
  const double value=log(x);
  if(isfinite(value)) { sum+=value; }
  }
return dx*sum;
}

double
Integrate_math(double xMin,
               double xMax,
               int stepCount,
               double (*fnct)(double))
{
const double dx=(xMax-xMin)/stepCount;
double sum=0.0, x=xMin+0.5*dx;
for(int i=0;i<stepCount;++i, x+=dx)
  {
  const double value=fnct(x);
  if(isfinite(value)) { sum+=value; }
  }
return dx*sum;
}
```

// ...(1/2)...

```
double                                                                          // ...(2/2)...
Integrate_param(double xMin,
                double xMax,
                int stepCount,
                ParamMathFnct fnct,
                void *mathParams)
{
const double dx=(xMax-xMin)/stepCount;
double sum=0.0, x=xMin+0.5*dx;
for(int i=0;i<stepCount;++i, x+=dx)
  {
  const double value=fnct(x,mathParams);
  if(isfinite(value)) { sum+=value; }
  }
return dx*sum;
}
```

12. L08_Perfs : Optimisation des performances

Motivation :

En introduction de cet enseignement, nous justifiions l'usage du langage C par sa bonne aptitude à produire du code exécutable efficace en terme de temps de calcul, et par extension en coup énergétique du service rendu (voir en 1.1). Nous argumentions en partie ceci par le fait que le compilateur est capable, lors de sa phase d'optimisation, de reformuler les traitements de manière minimale et parfaitement adaptée au matériel. Cette opération consiste principalement à retirer du code exécutable produit toutes les vérifications pour lesquelles il peut être prouvé à l'avance (par l'analyse des types notamment, mais pas seulement) qu'elles ne sont pas nécessaires lors de l'exécution et par la reformulation la plus concise possible du code utile résultant (voir en 1.2.1).

Seulement, nous n'avons pas encore cherché à bénéficier de cette possibilité. En effet, cet enseignement vise tout d'abord à découvrir les éléments fondamentaux du langage C et des recommandations raisonnables concernant sa mise en œuvre. L'objectif premier est donc la réalisation de programmes corrects et maintenables ; ce n'est qu'après l'atteinte de ces premières exigences que la question de l'optimisation doit se poser.

Consignes :

Au delà de la découverte très guidée de ces quelques nouvelles fonctionnalités, le travail demandé ici nécessitera d'être autonome dans la réutilisation des notions déjà acquises afin de les appliquer à un cas d'usage concret. Il faudra également savoir consulter la documentation. Veilliez à suivre scrupuleusement chacune des étapes en vous assurant à chaque fois d'avoir bien observé et compris ce qui vous est décrit.

12.1. Mise au point ou optimisation

Un fichier `GNUmakefile` générique a été mis à votre disposition dans le cadre de cet enseignement :

(http://www.enib.fr/~harrouet/Data/Courses/GNUmakefile)

Au début de celui-ci, il existe une variable `opt` positionnée à la valeur `0` par défaut. Comme indiqué dans les commentaires, cela signifie que nous ne souhaitons pas optimiser le code exécutable généré ; au contraire, nous voulons bénéficier de toute l'aide nécessaire à sa mise au point. Bien entendu, en positionnant cette variable `opt` à la valeur `1` nous pouvons exprimer le choix alternatif.

12.1.1. Permettre l'usage d'un débogueur

Le but premier de la mise au point d'un programme consiste à s'assurer du fait que les algorithmes exprimés effectuent bien le traitement tel qu'imaginé par le programmeur. Pour cela, il est assez courant d'avoir recours à un outil de débogage (comme nous l'avons vu en 2.3.2, en 2.3.5, en 4.4.4 et en 6.5) pour exécuter le code pas-à-pas, atteindre des points d'arrêt, inspecter la pile d'exécution et les variables... Il est donc indispensable que le code exécutable généré soit une traduction très fidèle, quasiment mot-à-mot, du code source. Dans ces conditions, chaque progression dans le code exécutable correspond à une progression facilement identifiable dans le code source.

En recherchant, dans le fichier *GNUmakefile* fourni, la prise en compte de la variable *opt* lorsqu'elle vaut *0*, vous devriez constater sans surprise que l'option *-g* sera indiquée au compilateur. Nous avons en effet déjà vu qu'elle provoque l'ajout aux fichiers objets produits, et donc au fichier exécutable qui les rassemblera, des informations permettant de faire le lien entre le code exécutable et le code source. De plus, l'option *-O0* indique au compilateur qu'aucune optimisation ne doit avoir lieu afin que le code exécutable généré soit très peu transformé par rapport au code source.

Vous avez déjà eu l'occasion d'expérimenter le débogueur dans ces conditions. À titre d'illustration, reprenez un des exercices pratiques précédents et recompilez-le en positionnant à la valeur *1* la variable *opt* du fichier *GNUmakefile*. Vous devriez constater que l'exécution dans le débogueur ne permet plus de faire le lien avec le code source.

12.1.2. Détecter au plus tôt les incohérences dans l'application

La prise en compte de la variable *opt* du fichier *GNUmakefile* lorsqu'elle vaut *0* introduit également l'option *-UNDEBUG*. Le préfixe *-U* est équivalent à la directive *#undef* du préprocesseur : dans le cas présent, nous supprimons donc la *macro* *NDEBUG* au cas où elle serait définie (par une option préalable par exemple). La présence ou l'absence de la *macro* *NDEBUG* (signifiant *no-debug*, pas de débogage) influence le comportement de la fonctionnalité *assert()*.

(http://en.cppreference.com/w/c/error/assert)

Il s'agit d'une *macro* définie dans le fichier d'en-tête standard *assert.h* dont le rôle est de terminer brutalement le programme avec *abort()* (précédé d'un message explicite) si l'expression qui lui est transmise est fausse. Seulement, cette *macro* se réduit à néant, et donc n'a aucun effet, si la *macro* *NDEBUG* est définie. L'intérêt est de tenter de détecter, pendant la phase de mise au point d'un programme (avec *NDEBUG* non définie donc), le plus grand nombre d'incohérences potentielles (des préconditions sur les paramètres d'une fonction par exemple). Ces vérifications peuvent être coûteuses en temps d'exécution, c'est pour cela qu'elles sont supprimées (avec *NDEBUG* définie) lorsque le programme est considéré comme suffisamment au point et que nous souhaitons en produire une version optimisée.

À titre d'illustration, reprenez un des exercices pratiques précédents, et faites usage de *assert()* avec une condition que vous savez être fausse. Observez alors le changement de comportement de votre programme selon qu'il est compilé pour la mise au point ou pour l'optimisation.

12.1.3. Instrumenter le code pour détecter les maladresses

Il existe des maladresses qui sont difficiles à détecter explicitement car elles peuvent être dues à une inattention ou une erreur de saisie. Elles peuvent provoquer, par exemple, la sortie d'un tableau lors de son parcours, l'usage d'un pointeur non initialisé, un mauvais usage de l'allocation dynamique, une fuite mémoire, le dépassement inattendu de la capacité d'un entier… Il est peu probable dans ces conditions que nous puissions utiliser *assert()* pour tenter de les détecter : il faudrait le faire absolument partout ! Heureusement, les compilateurs modernes offrent désormais des options pour produire automatiquement énormément de code qui tente de détecter de telles maladresses lors de l'exécution afin de nous les signaler au plus tôt.

La prise en compte de la variable *opt* du fichier *GNUmakefile* lorsqu'elle vaut *0* introduit également l'option *-fsanitize=address,undefined* sur les plateformes qui la supportent. La documentation du compilateur indique une grande variété d'options pour ces vérifications :

```
$ man gcc ↲
```

(c'est très long, monter/descendre avec les flèches, quitter avec la touche q)

À titre d'illustration, reprenez un des exercices pratiques précédents qui utilise un tableau (dynamique ou non), et modifiez temporairement un algorithme de parcours pour qu'il effectue une itération surnuméraire (<= à la place de < sur la condition de la boucle par exemple). Observez alors le changement de comportement de votre programme selon qu'il est compilé pour la mise au point ou pour l'optimisation. Lors de la mise au point, l'exécution s'arrête avec un message explicite au moment où notre programme tente un accès au delà du tableau. Au contraire, dans le cas d'un code optimisé cette maladresse passe inaperçue au moment où elle

est commise ; si elle doit avoir des conséquences, celles-ci ne seront visibles que bien plus tard et il sera difficile de remonter à l'origine du problème.

Toujours dans le but d'expérimenter, reprenez un exercice pratique qui fait usage de l'allocation dynamique de mémoire et retirez temporairement un appel à la libération d'un bloc mémoire inutile. Lors de la mise au point, à la fin de l'exécution du programme un message vous informe de cet oubli et vous indique même où l'allocation a eu lieu dans le code. Au contraire, dans le cas d'un code optimisé cette maladresse passe totalement inaperçue ; seul l'accroissement démesuré de la mémoire consommée par un tel programme qui commettrait cette maladresse de manière répétée pourrait éveiller notre attention.

Une toute autre maladresse que vous devez tester consiste à affecter à un entier une valeur proche de `INT_MAX` puis à l'incrémenter de telle façon qu'il devrait dépasser cette limite. Lors de la mise au point, un message d'avertissement est produit lorsque ce comportement indéfini (dépassement de la capacité d'un entier signé) est détecté. Au contraire, dans le cas d'un code optimisé cette maladresse passe totalement inaperçue ; seul l'aspect éventuellement aberrant des résultats de calcul du programme suite à ce dépassement de capacité pourrait éveiller notre attention.

12.1.4. Compiler pour l'optimisation

Tous les exercices qui vous ont été proposés dans le cadre de cet enseignement insistaient sur la nécessité d'écrire des programmes de test des fonctionnalités réalisées. Cette démarche doit être systématique et doit faire usage de tous les outils d'aide à la mise au point présentés ici. ***Ce n'est que lorsqu'on est absolument certain que ces fonctionnalités se comportent comme attendu qu'il est envisageable d'en accélérer l'exécution***. En effet, l'optimisation réalisée par le compilateur consiste à retirer du code produit toutes les vérifications désormais superflues pour ne laisser que les instructions qui sont strictement nécessaires aux traitements utiles. Ceci passe de plus par une reformulation profonde de ces derniers de telle façon qu'ils produisent le même effet observable mais selon un enchaînement d'opérations qui soit optimal pour le processeur. Dans ces conditions il n'y a quasiment plus moyen de faire le lien entre le code exécutable et le code source original, ce qui interdit toute démarche de mise au point.

La prise en compte de la variable `opt` du fichier `GNUmakefile` lorsqu'elle vaut `1` introduit l'option `-O3` qui demande au compilateur d'envisager les optimisations les plus agressives qu'il connaisse. L'option `-ffast-math` fait de même en autorisant, dans le but d'accélérer les calculs, une certaine souplesse dans l'usage des réels : négligence des arrondis, remplacement de certaines divisions par la multiplication de l'inverse, autorisation de la commutativité et de l'associativité qui sont correctes en arithmétique mais pas sur la représentation limitée en précision des réels... Bien entendu, l'option `-DNDEBUG` définit la macro `NDEBUG` (comme si la directive `#define` du préprocesseur était utilisée) qui rend inopérantes les invocations de `assert()`. L'option `-fomit-frame-pointer` permet l'économie de quelques opérations liées à l'usage du débogueur qui ne sera plus utilisé dorénavant. L'option `-march=native` indique au compilateur qu'il peut générer du code qui emploie toutes les instructions qui sont disponibles sur le processeur courant ; en faisant ceci, nous prenons le risque de produire un programme exécutable qui ne fonctionne qu'avec des processeurs identiques au processeur courant, en revanche il en exploite pleinement le jeu d'instructions.

Bien entendu, l'usage de ces options et de nombreuses autres est toujours discutable et dépendant du contexte applicatif visé ; il ne s'agit ici que d'exemples très classiques.

12.2. Recommandations pour l'optimisation

L'expérimentation autour des quelques recommandations suivantes s'appuiera sur une réalisation qui suivra à nouveau la démarche de programmation modulaire présentée au chapitre 2. Elle sera par exemple placée dans le répertoire `S3PRC_L08_Perfs`, contiendra un programme principal `prog08.c` et un module `perf` constitué d'un fichier de code source et de son fichier d'en-tête. La compilation utilisera le fichier `GNUmakefile` générique mis à votre disposition.

(http://www.enib.fr/~harrouet/Data/Courses/GNUmakefile)

Comme indiqué plus haut, ce n'est qu'après avoir validé le bon fonctionnement d'un programme que nous pouvons envisager l'analyse de ses performances et son optimisation.

Désormais, il faut compiler avec les options dédiées à l'optimisation comme indiqué en 12.1.4[138]. Toutefois, à chaque nouvelle adaptation du code source en vue d'obtenir de meilleures performances, il est nécessaire de revenir temporairement aux options de mise au point afin de s'assurer que le fonctionnement du programme reste correct.

12.2.1. Éliminer les répétitions inutiles

Une première précaution triviale à considérer, en vue d'accélérer l'exécution d'un traitement, consiste à éviter d'invoquer de manière répétée des opérations dont nous savons qu'elles produiront systématiquement le même effet.

Expérimentons ceci sur un exemple simpliste rendu volontairement très maladroit. La fonction suivante doit être fournie par le module *perf* :

```
int // number of letters in txt
Perf_alphaCount(const char *txt)
{
int count=0;
for(int i=0;i<(int)strlen(txt);++i) // ugly! many calls to strlen()
  { if(isalpha(txt[i])) { ++count; } }
return count;
}
```

Le programme principal contiendra ces fonctions :

```
#include <sys/time.h>

double // seconds since 1970/01/01 00:00:00 UTC
now(void)
{
struct timeval tv;
gettimeofday(&tv,NULL);
return (double)tv.tv_sec+1e-6*(double)tv.tv_usec;
}

#define TXT " This is a quite long text with many words.            \n" \
            " We aim at counting the number of letters it contains. \n"
#define TXT_x30 TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT \
                TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT
void
test_alphaCount(void)
{
printf("\n~~~~ %s() ~~~~\n",__func__);
const char *txt=TXT_x30;
const double t0=now();
double duration=0.0;
int iterCount;
for(iterCount=0;duration<2.0;++iterCount, duration=now()-t0)
  { Perf_alphaCount(txt); }
printf("%g it/s\n",iterCount/duration);
}
```

La fonction principale doit faire appel à la fonction *test_alphaCount()* afin que l'exécution de ce programme affiche les performances de la fonction *Perf_alphaCount()*. La mesure des performances repose sur un très grand nombre de répétitions du traitement à étudier, jusqu'à ce que la fonction *now()* nous indique une durée suffisamment longue (plus de deux secondes ici) pour qu'une mesure moyenne ait du sens.

138 Il y a encore quelques années ou décennies, il était courant d'utiliser des options permettant l'usage d'outils de profilage pour analyser les performances des applications. Les experts en optimisation considèrent désormais ceci contre-productif car le code ainsi instrumenté pour les mesures ne se comporte pas comme le code optimisé.

Compilez ce programme avec les options de mise au point et exécutez-le afin de vous assurer de son bon fonctionnement. Relevez alors plusieurs fois ses performances, et reprenez ceci avec les options d'optimisation : vous devez déjà constater un gain de performances énorme. Ceci met en évidence le fait qu'il n'est absolument pas pertinent de s'intéresser aux performances d'un traitement s'il n'est pas compilé avec les options d'optimisation.

Nous évoquions en introduction de cet exemple une maladresse grossière : en effet, l'évaluation de `strlen()` a lieu pour chaque caractère considéré dans la fonction `Perf_alphaCount()`. Or, en tant que concepteur de ce traitement, nous savons pertinemment que la longueur de la chaîne n'a aucune raison de changer d'une itération à l'autre. Il est donc inutile de réévaluer cette longueur pour chaque caractère visité ; de plus, `strlen()` est un traitement assez coûteux puisqu'il parcourt toute la chaîne jusqu'à détecter le caractère nul.

Modifiez alors la boucle de la fonction `Perf_alphaCount()` afin que `strlen()` ne soit invoquée qu'une seule fois et que son résultat soit mémorisé dans une variable. Vous pouvez conserver les deux versions en jouant sur des commentaires ou avec une compilation conditionnelle afin d'alterner entre ces deux solutions. Reprenez alors les tests et les mesures de performances avec cette nouvelle version. Vous devriez constater que le retrait de cette maladresse grossière est très bénéfique pour les performances, que ce soit en mise au point ou en optimisation.

Il ne s'agit ici que d'un cas particulièrement évident mais la démarche se généralise : ***à chaque fois qu'une boucle est réalisée, il faut s'interroger sur ce qui donne systématiquement le même résultat à chaque itération et le sortir de cette boucle en le mémorisant dans une variable qui sera réutilisée dans la boucle***. Il est à noter toutefois que les compilateurs modernes sont parfois capables de faire ce travail automatiquement lorsque le contenu de la boucle est facile à analyser. Dans le cas présent ce n'est pas le cas car le compilateur ne fait aucune supposition sur ce que fait la fonction `strlen()` ; il se garde donc de supposer que le résultat sera identique à chaque invocation.

12.2.2. La vision limitée du compilateur

La dernière remarque exprimée doit attirer votre attention sur le fait que le compilateur est capable de reformuler automatiquement un algorithme s'il est certain que la version modifiée produira exactement le même résultat que l'originale. S'il n'est pas en mesure de prouver cette équivalence, il adoptera au contraire une attitude très conservatrice dans le sens où sa première priorité est de produire un code exécutable qui aboutisse exactement au même résultat que ce qui est exprimé dans le code source. Ceci doit nous encourager à fournir au compilateur la meilleure vision possible du traitement réalisé par l'algorithme que nous exprimons.

Nous avons déjà rencontré en 7.2.3 la définition de fonctions qualifiées de `inline static`, ce qui correspond exactement à cette attente : puisque le compilateur voit le code de la fonction appelée à l'endroit de son appel, il sait exactement ce que la fonction fait et peut en déduire les interactions potentielles avec les données du contexte appelant.

```
int *p=obtainAnAddress();
int a=(*p)*4;
doSomething();
int b=(*p)+3;
```

Lors de l'analyse du code de cet exemple, si la définition de la fonction `doSomething()` n'est pas visible, le compilateur n'est pas en mesure de prouver que la donnée située à l'adresse désignée par le pointeur `p` ne sera pas modifiée par cet appel. Il y aura donc un premier accès à cette adresse pour calculer la variable `a` puis, plus tard, un second accès pour calculer la variable `b`.

Supposons maintenant que la définition de la fonction `doSomething()` soit visible et qu'elle n'ait aucune influence sur ce qui est désigné par le pointeur `p`. Dans ces nouvelles conditions, le compilateur pourrait prendre la liberté de reformuler le code précédent de cette façon :

```
int *p=obtainAnAddress();
const int tmp=(*p);
int a=tmp*4;
doSomething();
int b=tmp+3;
```

Selon cette nouvelle formulation, un seul accès à l'adresse indiquée par le pointeur *p* a lieu et la valeur *tmp* est directement réutilisée dans les calculs de *a* et *b*, ce qui se révèle probablement plus efficace[139].

Il pourrait alors être tentant de qualifier toutes les fonctions de *inline static* mais ce serait contre-productif. En effet, le code de ces fonctions est dupliqué dans le programme exécutable à chaque invocation, ce qui produit un code beaucoup plus volumineux, et la taille du code peut avoir une influence très négative sur les performances[140]. Nous réservons donc le qualificatif *inline static* aux fonctions qui ne sont constituées que de très peu d'opérations.

12.2.3. Expliciter nos intentions

Toutefois, toujours dans le contexte de notre exemple précédent, même si la définition de la fonction *doSomething()* n'est pas visible du compilateur lors de son appel, le développeur, lui, sait certainement très bien pourquoi il invoque cette fonction et quel en est l'effet attendu. Il sait donc parfaitement si cette invocation est susceptible de modifier ce qui se trouve à l'adresse désignée par le pointeur *p* ou non. Dans ce dernier cas (le plus probable), c'est donc au développeur de formuler l'algorithme en levant explicitement l'ambiguïté par l'usage d'une variable (*tmp* ici) qui mémorise le résultat d'un unique accès à l'adresse indiquée (par *p* ici).

La démarche illustrée par cet exemple doit systématiquement être généralisée : ***il faut rendre explicite au compilateur ce que nous savons mais qu'il ne peut déduire de la simple analyse du code visible localement***[141]. Ceci passe notamment par l'usage de variables pour mémoriser des résultats plutôt que de recourir à de multiples évaluations qui donneraient des résultats identiques, ce qui rejoint la recommandation formulée en 12.2.1. Cette recommandation est particulièrement pertinente lorsque des pointeurs sont utilisés, car le compilateur a généralement une mauvaise visibilité sur ce qui est susceptible d'interagir avec la donnée désignée. La règle suivante sera alors appliquée : ***nous nous interdisons d'accéder plusieurs fois à la même adresse avec un pointeur si la valeur désignée ne doit pas être différente***.

De la même façon, ***il faut éviter de réutiliser la même variable pour des calculs qui sont indépendants***. Il n'y a en effet rien à gagner d'une telle économie supposée de variable[142] : les variables ne représentent qu'une abstraction pour exprimer les algorithmes. Elles n'ont, pour la plupart, pas d'emplacement en mémoire (sauf si on cherche explicitement à en connaître l'adresse) car l'optimisation par le compilateur peut les faire disparaître (au profit d'une mémorisation dans les registres du processeur) ou au contraire en créer de nouvelles pour stocker les résultats intermédiaires de longs calculs. La réutilisation d'une même variable risque de créer artificiellement une dépendance entre des calculs qui sont en réalité indépendants. Dans ces conditions, le compilateur s'interdira de remanier ces calculs de manière plus efficace si, lorsque l'algorithme n'est pas trivial, il ne peut prouver que les résultats ne s'en trouveront pas modifiés. Si au contraire, nous prenons soin d'utiliser des variables distinctes pour des calculs distincts, nous n'introduisons pas de dépendances artificielles entre ces calculs et le compilateur aura toute l'opportunité de les réorganiser de manière plus efficace.

---

139 Vous apprendrez dans les matières S5-MIP et S6-MIP de l'ENIB que les accès à la mémoire sont bien plus coûteux en temps d'exécution que la réutilisation des registres d'un processeur.

140 Ceci est lié à l'épuisement de la mémoire cache que le processeur dédie aux instructions ; cet aspect est abordé dans la matière S7-SEN de l'ENIB.

141 Il existe toutefois des options d'optimisation qui repoussent les décisions d'optimisation à l'édition de liens (comme si tout le code tenait dans une unique unité de compilation) mais ceci sort du cadre de cet enseignement.

142 C'est une erreur très courante que commettent les programmeurs débutants.

Pour aller plus loin dans l'explicitation de nos intentions, il est recommandé de qualifier avec `const` les variables pour lesquelles nous savons qu'aucune modification ne doit avoir lieu (pas seulement les données désignées par des pointeurs comme indiqué en 3.4.5). Cette information peut aider le compilateur dans son analyse du code et lui offrir plus de latitudes dans les reformulations qu'il peut envisager.

12.2.4. Préférer les entiers signés aux entiers non-signés

Nous avons évoqué, en 9.6, le fait que la représentation en complément-à-deux permettait de réaliser les opérations arithmétiques sur les entiers signés de la même façon que sur les entiers non-signés. Il est donc peu probable que l'exécution de ces opérations aient des performances différentes dans le cas signé ou non-signé. Cependant, nous avons également indiqué en 4.3.1, que le débordement de capacité sur un entier non-signé avait un comportement parfaitement défini (le rebouclage vers l'autre extrémité de la plage de valeurs), alors que ce comportement est indéfini sur un entier signé. C'est la distinction entre ces comportements défini ou indéfini qui peut avoir une influence sur l'optimisation des algorithmes, notamment lorsque ces entiers servent d'indices pour accéder à des données.

Expérimentons ceci en ajoutant au module `perf` cette fonction qui réalise dans `cMtx` le produit des matrices `aMtx` et `bMtx` (de côté `size`) :

```
void
Perf_mtxProduct(const double *aMtx,
                const double *bMtx,
                double *cMtx,
                index_t size)
{
for(index_t i=0;i<size;++i)
  {
  for(index_t j=0;j<size;++j)
    {
    const index_t cIdx=i*size+j;
    cMtx[cIdx]=0.0;
    for(index_t k=0;k<size;++k)
      {
      const index_t aIdx=i*size+k, bIdx=k*size+j;
      cMtx[cIdx]+=aMtx[aIdx]*bMtx[bIdx]; // ugly! many stores to cMtx[cIdx]
      }
    }
  }
}
```

Le type `index_t` est inconnu et vous devrez, dans un premier temps, indiquer dans le fichier d'en-tête qu'il est équivalent au type `unsigned int`. Dans le programme principal, vous réaliserez une fonction `test_mtxProduct()` qui reprendra le même principe que la fonction `test_alphaCount()` pour mesurer les performances de la fonction `Perf_mtxProduct()` sur trois matrices de côté `100` (des tableaux de taille `100*100` donc) et qui sera appelée par la fonction principale. Après en avoir testé le fonctionnement en l'ayant compilé pour la mise au point, fabriquez à nouveau ce programme pour l'optimisation et relevez plusieurs fois les performances reportées.

Avant d'aborder la problématique du type `index_t` artificiellement introduit ici, intéressons nous au commentaire sur la maladresse introduite dans la fonction `Perf_mtxProduct()`. Cette formulation va effectivement à l'encontre de la règle que nous énoncions en 12.2.3 concernant l'usage répété d'un pointeur désignant la même donnée : à chaque itération selon l'indice `k`, nous modifions le même élément de `cMtx`. Adaptez alors cette boucle en faisant usage d'une variable qui accumule les incrémentations successives pour finir par une écriture unique du résultat sur l'élément de `cMtx` concerné. Utilisez des commentaires ou la compilation conditionnelle pour alterner entre ces deux formulations. Après en avoir testé le bon

fonctionnement, relevez à plusieurs reprises les performances de cette formulation alternative ; vous devriez constater une certaine amélioration pour ces dernières.

À ce propos, dans sa version C99 le langage C introduit le mot clef `restrict` qui, associé à un pointeur, indique au compilateur que les données accessibles par ce pointeur ne le seront que par celui-ci au sein de l'algorithme courant.

(http://en.cppreference.com/w/c/language/restrict)

Fort de cette information, le compilateur peut s'autoriser des reformulations qui ignorent les interactions potentielles entre les différents pointeurs du problème ; ceci est assez proche de la préoccupation que nous avions en introduisant explicitement la variable d'accumulation dans notre exemple.

Intervenez maintenant sur le type `index_t` dans le fichier d'en-tête (avec des commentaires ou la compilation conditionnelle) afin qu'il soit dorénavant équivalent au type `int` ; il était non-signé jusqu'alors et devient désormais signé. Reprenez les mesures de performances avec et sans la maladresse liée à l'absence de la variable d'accumulation discutée précédemment. Dans ces nouvelles conditions vous devriez constater un nouvel accroissement des performances, et notamment de manière très significative lorsque la variable d'accumulation est utilisée.

L'explication repose uniquement sur le fait que, comme indiqué en 4.3.1, le débordement de capacité sur un entier signé donne un comportement indéfini : un algorithme qui utilise des indices signés ne peut en aucune façon être considéré comme correct si ces indices subissent un débordement de capacité. Par conséquent, à l'analyse du code, le compilateur peut supposer que l'incrémentation d'un indice signé ne pourra jamais donner un indice inférieur : le programme aurait alors un comportement indéfini. Avec des indices non-signés, il serait au contraire tout à fait légitime de s'appuyer sur le fait que les indices trop grands rebouclent vers zéro et le compilateur ne pourrait donc pas supposer impossible qu'un indice devienne inférieur suite à son incrémentation. Cette propriété fait que dans le cas d'indices signés le compilateur peut reformuler le code en supposant qu'un tel rebouclage ne peut pas légitimement se produire alors qu'il s'interdira de faire cette supposition (même si dans les faits le cas ne se présente pas) avec des indices non-signés.

Au bout du compte nous constatons une reformulation bien plus efficace de l'algorithme par le compilateur, non pas parce que les opérations arithmétiques sur les entiers signés sont plus rapides (elles ne le sont absolument pas !), mais parce que le compilateur se débarrasse de précautions superflues lorsqu'il organise les accès à la mémoire. La règle suivante sera désormais appliquée : ***nous utilisons systématiquement des entiers signés sauf dans quelques cas très rares où l'usage d'entiers non-signés représente un aspect essentiel du problème*** (pour les opérations bit-à-bit par exemple)[143].

12.3. Résumé

Ici s'achève cette huitième séance pratique dans laquelle nous avons expérimenté les capacités d'optimisation du compilateur en vue d'améliorer les performances des programmes réalisés.

Il en ressort que la démarche de mise au point et la démarche d'optimisation sont deux phases bien distinctes et exclusives l'une de l'autre. La mise au point doit être la priorité afin de s'assurer que les traitements exprimés dans les algorithmes produisent bien les effets attendus sans introduire d'indétermination. Ce n'est qu'après ces vérifications effectuées que l'optimisation doit être abordée, car elle provoque un remaniement du code exécutable tellement profond qu'il compromet toute tentative d'usage des outils de mise au point.

143 Les principaux concepteurs du standard C++ reconnaissent notamment que l'usage d'entiers non-signés dans les bibliothèques standards (C et C++) est une grossière erreur commise par le passé et qui ne subsiste qu'afin d'assurer la compatibilité avec du code ancien (voir la session *« Interactive Panel: Ask Us Anything »* de la conférence Going Native 2013 facilement accessible en ligne).

Indépendamment de la formulation précise des algorithmes, il faut savoir que certaines opérations sont réputées comme très coûteuses en temps d'exécution, nous éviterons donc de les invoquer de manière intensive (en boucle par exemple). Parmi les plus courantes il y a :

- les entrées/sorties (lectures/écritures dans des fichiers ou le terminal par exemple) : il faut préférer quelques transferts volumineux à de multiples petits transferts,
- l'allocation dynamique de mémoire : il vaut mieux allouer rarement et réutiliser longuement la mémoire obtenue plutôt que de répéter les allocations et les libérations.

Les précautions à prendre pour améliorer les performances lors de la formulation des algorithmes peuvent se résumer à exprimer le plus simplement possible nos intentions au compilateur sans introduire de contraintes qui ne soient pas justifiées dans le problème traité. Il ne s'agit, à proprement parler, pas d'optimiser mais de laisser au compilateur l'opportunité de le faire. En première approche nous pouvons nous contenter de :

- sortir des boucles les traitements dont nous savons qu'ils donneront systématiquement le même résultat à chaque itération,
- utiliser le qualificatif `inline static` pour les fonctions courtes qui sont invoquées de manière répétée,
- ne pas accéder plusieurs fois à la même adresse avec un pointeur si la valeur désignée ne doit pas être différente,
- ne pas hésiter à utiliser de multiples variables et éviter de réutiliser la même variable pour des calculs qui sont indépendants,
- qualifier avec `const` les variables pour lesquelles nous savons qu'aucune modification ne doit avoir lieu,
- utiliser systématiquement des entiers signés sauf dans quelques cas très rares où l'usage d'entiers non-signés représente un aspect essentiel du problème.

Il se peut cependant que, sur des algorithmes suffisamment simples (parcours monotone de tableaux unidimensionnels par exemple), un compilateur moderne soit capable de prouver que les cas problématiques redoutés ne se produiront pas. Il se peut également qu'il puisse se donner les moyens de les détecter à l'exécution et générer plusieurs versions du même algorithme qui seront invoquées selon les plages d'adresses et d'indices rencontrés lors de l'exécution. Dans ces conditions très favorables, les précautions que nous prenons pourraient nous sembler inutiles car redondantes avec celles prises par le compilateur.

Toutefois, il convient de toujours s'efforcer d'appliquer ces précautions car tous les compilateurs ne sont pas aussi performants dans leur analyse des interactions potentielles entre les données d'un problème. De plus, même lorsqu'un compilateur est capable de détecter les cas favorables d'un algorithme, une modification mineure de ce dernier peut mettre soudainement en échec cette analyse.

Pour en revenir aux motivations de l'usage du langage C exprimées en 1.1, si nous choisissons d'utiliser ce langage c'est que nous avons des attentes en matière de performances des traitements réalisés. Il convient donc de se donner tous les moyens de parvenir à cet objectif en appliquant scrupuleusement les recommandations qui sont formulées ici[144] ; il s'agit d'un travail d'attention incessant qui demande de la rigueur dans l'application de règles systématiques car la moindre maladresse peut ruiner de nombreux efforts.

144 Il existe bien entendu des axes d'optimisation supplémentaires. Ceux-ci sont liés à une bonne connaissance du fonctionnement des processeurs, des machines et des systèmes informatiques et ne pourront être abordés qu'après avoir suivi les enseignements des matières S5-MIP, S6-MIP, S7-SEN et S7-CRS de l'ENIB.

Voici le contenu des fichiers réalisés dans ce sujet, tels qu'ils doivent être si vous avez suivi scrupuleusement les recommandations qui vous ont été données à chaque étape.

```
//---- prog08.c ----
#include <stdio.h>
#include <sys/time.h>
#include "perf.h"

double // seconds since 1970/01/01 00:00:00 UTC
now(void)
{
struct timeval tv;
gettimeofday(&tv,NULL);
return (double)tv.tv_sec+1e-6*(double)tv.tv_usec;
}

#define TXT " This is a quite long text with many words.            \n" \
            " We aim at counting the number of letters it contains. \n"
#define TXT_x30 TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT \
                TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT TXT

void
test_alphaCount(void)
{
printf("\n~~~~ %s() ~~~~\n",__func__);
printf("REPEAT_STRLEN=%d\n",REPEAT_STRLEN);
const char *txt=TXT_x30;
const double t0=now();
double duration=0.0;
int iterCount;
for(iterCount=0;duration<2.0;++iterCount, duration=now()-t0)
  { Perf_alphaCount(txt); }
printf("%g it/s\n",iterCount/duration);
}

#define MTX_SIZE 100

void
test_mtxProduct(void)
{
printf("\n~~~~ %s() ~~~~\n",__func__);
printf("USE_UNSIGNED_INDEX=%d\n",USE_UNSIGNED_INDEX);
printf("USE_ACCUM=%d\n",USE_ACCUM);
double aMtx[MTX_SIZE*MTX_SIZE];
double bMtx[MTX_SIZE*MTX_SIZE];
double cMtx[MTX_SIZE*MTX_SIZE];
const double t0=now();
double duration=0.0;
int iterCount;
for(iterCount=0;duration<2.0;++iterCount, duration=now()-t0)
  { Perf_mtxProduct(aMtx,bMtx,cMtx,MTX_SIZE); }
printf("%g it/s\n",iterCount/duration);
}



                                                                  // ...(1/2)...
```

```
int                                                                    // ...(2/2)...
main(void)
{
test_alphaCount();
test_mtxProduct();
return 0;
}
```

```
//---- perf.h ----
#ifndef PERF_H
#define PERF_H 1

#define REPEAT_STRLEN 0

int // number of letters in txt
Perf_alphaCount(const char *txt);

#define USE_UNSIGNED_INDEX 0
#define USE_ACCUM          1

#if USE_UNSIGNED_INDEX
  typedef unsigned int index_t;
#else
  typedef int index_t;
#endif

void
Perf_mtxProduct(const double *aMtx,
                const double *bMtx,
                double *cMtx,
                index_t size);

#endif // PERF_H
```

```
//---- perf.c ----
#include "perf.h"
#include <string.h>
#include <ctype.h>

int // number of letters in txt
Perf_alphaCount(const char *txt)
{
int count=0;
#if REPEAT_STRLEN
  for(int i=0;i<(int)strlen(txt);++i) // ugly! many calls to strlen()
#else
  for(int i=0, len=(int)strlen(txt);i<len;++i)
#endif
    { if(isalpha(txt[i])) { ++count; } }
return count;
}



                                                                       // ...(1/2)...
```

```
void                                                                    // ...(2/2)...
Perf_mtxProduct(const double *aMtx,
                const double *bMtx,
                double *cMtx,
                index_t size)
{
for(index_t i=0;i<size;++i)
  {
  for(index_t j=0;j<size;++j)
    {
    const index_t cIdx=i*size+j;
#if USE_ACCUM
    double accum=0.0;
    for(index_t k=0;k<size;++k)
      {
      const index_t aIdx=i*size+k, bIdx=k*size+j;
      accum+=aMtx[aIdx]*bMtx[bIdx];
      }
    cMtx[cIdx]=accum;
#else
    cMtx[cIdx]=0.0;
    for(index_t k=0;k<size;++k)
      {
      const index_t aIdx=i*size+k, bIdx=k*size+j;
      cMtx[cIdx]+=aMtx[aIdx]*bMtx[bIdx]; // ugly! many stores to cMtx[cIdx]
      }
#endif
    }
  }
}
```

Glossaire

Nous retrouvons ici les termes de vocabulaire qui sont couramment utilisés dans le contexte du langage C. Lorsqu'ils sont introduits pour la première fois dans le texte de ce document, ils sont marqués du symbole ***(vocabulaire)***.

adresse d'une donnée
: L'emplacement où se situe une donnée dans la mémoire. Les adresses sont manipulées avec les types pointeurs.

alignement
: La propriété d'un type qui contraint l'adresse qu'une donnée peut occuper en mémoire. Les propriétés d'alignement des différents types varient d'une plateforme d'exécution à une autre. Le compilateur en a connaissance et choisit le placement des données en conséquence.

allocation dynamique de mémoire
: Le procédé qui consiste à réclamer explicitement au système d'exploitation un bloc d'octets de taille choisie afin que le programme en fasse l'usage qui lui convient. Un tel bloc devra être explicitement libéré.

alternative multiple
: Une construction du langage qui conditionne l'exécution d'un nombre quelconque de portions de code à l'évaluation unique de la valeur entière d'une expression. Les mots-clefs `switch` , `case` et `default` matérialisent cette alternative.

alternative simple
: Une construction du langage qui conditionne l'exécution d'une ou deux portions de code à la valeur logique d'une expression. Les mots-clefs `if` et `else` matérialisent cette alternative.

ASCII
: Une table de conversion standard entre des codes numériques et des caractères textuels.

bibliothèque standard
: L'ensemble des fonctionnalités dédiées à des tâches usuelles, déjà compilées et livrées telles quelles lors de l'installation du compilateur sur le système d'exploitation. Tout programme a implicitement accès à ces fonctionnalités, à condition d'inclure les fichiers d'en-tête standards.

bit
: Un chiffre binaire (*binary-digit*). Un *bit* ne peut valoir que `0` ou `1` et est souvent interprété comme une puissance de deux selon sa position dans une séquence. Un octet est constitué de huit *bits*.

bit-à-bit
: Une opération sur des entiers qui ne les considère pas pour leur valeur arithmétique mais qui effectue des opérations logiques sur les *bits* qui les constituent.

bloc
Une paire d'accolades contenant des déclarations et des instructions. Les blocs de code limitent la portée des identificateurs qui y sont déclarés.

boucle à pré-condition
Une construction du langage qui conditionne la répétition d'une portion de code à la valeur logique d'une expression qui est évaluée avant chaque nouvelle itération. Le mot-clef `while` matérialise cette boucle.

boucle à post-condition
Une construction du langage qui conditionne la répétition d'une portion de code à la valeur logique d'une expression qui est évaluée après chaque nouvelle itération. Les mot-clefs `do` et `while` matérialisent cette boucle.

boucle avec compteur
Une construction du langage qui détermine la répétition d'une portion de code selon un compteur. Le mot-clef `for` matérialise cette boucle (mais il permet d'exprimer d'autres formes de boucles).

byte
La plus petite entité adressable (que l'on peut désigner pour y lire ou y écrire) dans la mémoire. Dans la pratique, bien que rien ne l'impose, il s'agit généralement d'un octet.

caractère nul
Le caractère qui est placé à la fin d'une chaîne pour en marquer la fin. Son code ASCII vaut zéro et il est usuellement représenté par la constante littérale `'\0'`.

cast
Le terme anglais usuellement employé pour l'opération de coercition.

chaîne de caractères
Un tableau contenant une séquence de caractères terminée par le caractère nul. Une chaîne sert à représenter du texte.

classe de stockage automatique
Une propriété des variables qui fait qu'elles apparaissent automatiquement à l'entrée dans un bloc de code (une fonction par exemple) et qu'elles disparaissent à sa sortie. Les variables automatiques ont une valeur initiale indéterminée si on ne les initialise pas explicitement.

classe de stockage statique
Une propriété des variables qui fait qu'elles existent pendant toute la durée du programme. Les variables globales (déclarées à l'extérieur de toute fonction) et les variables à portée locales mais précédées du mot-clef `static` ont une classe de stockage statique et ont une valeur initiale nulle si on ne les initialise pas explicitement.

cible à fabriquer
Ce qui doit être produit par une règle de fabrication d'un fichier `makefile`. Il s'agit généralement d'un fichier objet ou d'un programme exécutable.

coercition
L'action de convertir explicitement un type de donnée vers un autre. Sa forme consiste à faire précéder une expression par le nom du type visé entre parenthèses.

commande de fabrication
La commande spécifiée dans un fichier `makefile` qui doit être invoquée pour générer une cible à partir de ses prérequis. Il s'agit généralement de provoquer une compilation ou une édition de liens.

compilateur
L'outil informatique chargé de réaliser la compilation. Les outils `gcc` ou `cl.exe` par exemple sont des compilateurs.

compilation
L'action de traduire un code source exprimé en langage C vers un format exécutable que le système d'exploitation et la machine informatique comprennent.

compilation conditionnelle
Une construction à base de directives du préprocesseur faisant en sorte qu'une partie du code source soit prise en compte ou bien ignorée. Un usage très courant consiste à proposer pour une même fonctionnalité plusieurs formulations qui concernent chacune une plateforme d'exécution différente.

compilation séparée
La démarche qui consiste à répartir le code source d'un programme en de multiples modules compilés séparément. Pour une application conséquente, cela permet un gain de temps sur sa fabrication en ne recompilant que les modules qui ont été modifiés.

complément-à-deux
La convention pour représenter les valeurs négatives des entiers qui permet de leur appliquer les opérations arithmétiques usuelles de la même façon qu'avec les valeurs entières positives.

condition
Une expression dont on ne retient que la valeur logique pour prendre une décision. En langage C toute valeur nulle (quel que soit son type) est considérée comme fausse et toute autre valeur non nulle (quelle qu'elle soit) est considérée comme vraie.

contexte appelant
L'ensemble des données qui étaient accessibles juste avant l'appel à une fonction. Il n'est pas directement accessible depuis le contexte de la fonction appelée.

déclaration d'une fonction
L'écriture du prototype d'une fonction suivi d'un point-virgule. À défaut d'une définition, une telle déclaration doit être connue du compilateur afin qu'un appel à cette fonction puisse être exprimé.

définition d'une fonction
L'écriture du prototype d'une fonction suivi d'un bloc de code qui constitue son algorithme. Un programme n'est pas complet s'il contient des appels à des fonctions qui ne sont pas définies.

débogueur
L'outil informatique qui permet d'inspecter un programme pendant son exécution. Il sert principalement à investiguer lorsqu'une erreur se manifeste pendant le fonctionnement du programme. L'outil `gdb` est un débogueur ; il existe de multiples interfaces graphiques pour en présenter les informations.

déréférencement
L'action d'accéder à la donnée désignée par un pointeur. L'intention peut être de la modifier ou de la consulter. Le symbole `*` matérialise cette opération.

directive
Une construction du code source destinée à être interprétée par le préprocesseur. Les directives commencent par le symbole `#`.

éditeur de liens
L'outil informatique chargé de réaliser l'édition de liens. Les outils `ld` ou `link.exe` par exemple sont des éditeurs de lien. Cet outil est généralement invoqué automatiquement par le compilateur.

édition de liens
La dernière étape de la fabrication d'un programme exécutable qui consiste à rassembler les modules compilés séparément. Cette étape vérifie notamment que les multiples modules ne définissent qu'une seule fois chaque fonction du programme et qu'il n'en manque pas.

effet de bord
L'effet qui peut être produit par l'évaluation d'une expression mais qui n'est pas son résultat. L'affichage d'un message ou la modification d'une donnée en mémoire sont des effets de bord.

encapsulation
La démarche qui consiste à définir une structure de données et l'ensemble des fonctions qui décrivent le jeu d'opérations usuelles qui ont du sens pour ce type. Les fonctions en question forment l'unique interface de programmation pour ce type.

erreur de segmentation
L'erreur qui se produit lorsqu'un programme tente d'utiliser une zone de la mémoire qui lui est inaccessible. Une telle erreur est émise par le système d'exploitation et termine brutalement le programme.

espace d'adressage
L'ensemble des adresses qui peuvent être désignées par un pointeur. L'étendue de la mémoire effectivement disponible sur une plateforme d'exécution est généralement bien moins vaste.

expression
Une formulation dont l'évaluation doit produire un résultat. En langage C il n'y a pas de distinction entre les expressions arithmétiques et logiques.

fichier d'en-tête
Un fichier contenant essentiellement des déclarations de types et de fonctions. Dans le cadre de la démarche de compilation séparée, de tels fichiers sont inclus dans le code source afin d'avoir accès aux déclarations de fonctions qui sont définies dans d'autres modules. Le nom d'un fichier d'en-tête a généralement une extension `.h`.

fichier d'en-tête standard
Un fichier d'en-tête qui est livré avec la bibliothèque standard à l'installation du compilateur. Il en existe de nombreux ; ce document exploite ceux-ci :

`ctype.h`	*character types*
`float.h`	*floating types*
`limits.h`	*implementation-defined constants*
`math.h`	*mathematical declarations*
`stdbool.h`	*boolean type and values*
`stddef.h`	*standard type definitions*
`stdint.h`	*integer types*
`stdio.h`	*standard buffered input/output*
`stdlib.h`	*standard library definitions*
`string.h`	*string operations*

fichier makefile
Un fichier décrivant toutes les étapes permettant d'obtenir un programme exécutable à partir de ses fichiers de code source. Un tel fichier est exploité par la commande `make`.

fichier objet
Le fichier qui est obtenu en compilant le code source d'un module mais sans aller jusqu'à l'édition de liens. L'éditeur de lien rassemble les multiples fichiers objets pour produire le programme exécutable.

flux
Un objet qui est relié à un moyen de communication (fichier, terminal…), qui permet d'en connaître l'état et qui propose des fonctionnalités de formatage. Les fichiers et les entrées-sorties standards sont généralement manipulés par des flux.

fonction principale
La fonction qui est automatiquement appelée lorsque le programme démarre et qui met fin au programme lorsqu'elle se termine ; c'est le point d'entrée du programme. La fonction `main()` est la fonction principale d'un programme en langage C.

fuite mémoire
: Le fait qu'un programme ne libère pas des blocs mémoire qu'il a obtenus dynamiquement bien qu'il n'en ait plus besoin. Cette situation peut causer l'épuisement de la mémoire disponible dans le système.

identificateur
: Le nom qui est donné à quelque chose d'identifiable dans le langage. Ce nom peut désigner une fonction, une variable, un type…

indexation
: L'opération qui consiste à accéder à un élément d'un tableau en spécifiant un indice. L'intention peut être de le modifier ou de la consulter. Les symboles `[` et `]` matérialisent cette opération.

indice d'un tableau
: Un entier utilisé pour désigner un élément parmi tous ceux d'un tableau.

indirection
: La pratique qui consiste à ne pas manipuler directement une donnée par une variable mais à travers un pointeur. De multiples niveaux d'indirection peuvent être imbriqués.

instruction
: La moindre action qui puisse être exécutée. Un programme est une séquence d'instructions.

libération de mémoire dynamique
: L'action qui consiste à rendre au système d'exploitation un bloc d'octets obtenu par allocation dynamique. Un programme doit libérer un tel bloc dès qu'il n'en a plus besoin afin qu'il redevienne disponible pour un autre usage dans le système.

liste d'initialisation d'un tableau
: La construction qui permet d'énumérer, lors de sa déclaration, les valeurs que doit contenir un tableau. Les symboles `{` et `,` et `}` matérialisent cette construction.

lvalue
: Une expression que l'on peut retrouver comme membre gauche d'une opération d'affectation afin d'accueillir un résultat. C'est le cas d'une simple variable, du déréférencement d'un pointeur ou encore d'un élément d'un tableau.

macro-instruction
: Une directive du préprocesseur qui substitue dans le code source un unique mot (éventuellement accompagné de paramètres) par un texte quelconque. Les *macros* (en raccourci) servent souvent à nommer des constantes littérales.

masque binaire
: Une valeur entière dont les *bits* sont positionnés explicitement afin de la combiner avec une autre valeur entière dans une opération *bit-à-bit*. Ceci permet notamment d'isoler quelques *bits* de cette autre valeur, d'en forcer à `0` ou bien à `1`.

membre d'une structure
: Une des variables constitutives d'une structure de données. La structure peut être considérée comme un tout, ou bien chacun de ses membres peut être manipulé séparément.

module
: Un fichier de code source réalisant des fonctionnalités sur un même thème. De multiples modules constituent une application ; ils sont compilés séparément.

octet
: Un ensemble de huit *bits*. Un octet peut être vu comme un petit entier permettant de distinguer deux-cent-cinquante-six valeurs différentes.

optimisation
L'étape de la compilation qui consiste à simplifier et réordonner les instructions que la machine informatique devra exécuter afin que l'exécution d'un traitement soit la plus rapide possible tout en restant conforme aux algorithmes exprimés dans le code source.

opérateur conditionnel
La dénomination alternative pour l'opérateur ternaire.

opérateur ternaire
Un opérateur composé de trois membres : une condition, une expression à évaluer si la condition est vérifiée, une autre expression à évaluer sinon. Le résultat de cet opérateur est le résultat d'une de ses deux expressions. Les symboles `?` et `:` matérialisent cet opérateur.

paramètre effectif
La valeur qui est effectivement transmise lors de l'appel d'une fonction. C'est le résultat de l'évaluation d'une expression.

paramètre formel
Le nom et le type d'un paramètre qui est attendu par une fonction. Il est initialisé à l'entrée de la fonction par une copie du paramètre formel.

pas-à-pas
La démarche qui consiste à suspendre l'exécution du programme après chaque ligne de code lors d'une session de débogage. Cela permet d'investiguer les conditions de réalisation et les effets de chacune d'elles lors de la recherche d'une erreur d'exécution.

passage par adresse
La pratique qui consiste à utiliser un pointeur comme paramètre formel d'une fonction et à fournir comme paramètre effectif l'adresse d'une donnée afin qu'elle puisse être modifiée. L'adresse en question est bien transmise par valeur ; ce n'est qu'une émulation d'un passage par référence.

passage par référence
Le fait que le paramètre formel d'une fonction soit directement lié au paramètre effectif et peut modifier ce dernier. Ce mode de passage de paramètre n'existe pas en langage C et doit être émulé par un passage par adresse.

passage par valeur
Le fait que le paramètre formel d'une fonction ne manipule qu'une copie du paramètre effectif et ne peut pas modifier ce dernier. C'est l'unique mode de passage de paramètre en langage C .

pile d'exécution
La zone de l'espace d'adressage qui permet de contenir les données associées aux appels de fonctions. On y trouve notamment les paramètres et les variables de classe de stockage automatique des fonctions.

point d'arrêt
Une ligne de code désignée dans un débogueur pour que l'exécution soit suspendue lorsqu'elle est atteinte. Un point-d'arrêt permet d'atteindre rapidement une zone de code digne d'intérêt lors d'une session de débogage.

pointeur
Un type de donnée qui permet de manipuler l'adresse d'une autre donnée. Le type d'un pointeur détermine l'interprétation de la donnée qui est située à cette adresse.

pointeur nul
Un pointeur qui mémorise l'adresse `0` afin de signifier qu'il ne désigne aucune donnée utile. La *macro* `NULL` peut être utilisée pour une telle initialisation.

portée globale
: Une propriété des identificateurs qui fait que leur visibilité est étendue à l'ensemble du code. Tout ce qui est déclaré à l'extérieur de toute fonction (variable, type, fonction...) a une portée globale.

portée locale
: Une propriété des identificateurs qui fait que leur visibilité est limitée au bloc de code englobant. Tout ce qui est déclaré à l'intérieur d'une fonction (variable, type...) a une portée locale.

préprocesseur
: L'étape de la compilation qui consiste à modifier à la volée le code source que le compilateur analysera effectivement. Des directives permettent de spécifier ces transformations lexicales.

prérequis à la fabrication
: Ce qui est nécessaire à la production d'une cible par une règle de fabrication d'un fichier `makefile`. Dans le cas où la cible est un fichier objet, il s'agit d'un fichier de code source et des fichier d'en-tête qu'il inclut. Dans le cas où la cible est un programme exécutable, il s'agit de l'ensemble des fichiers objets dont il doit être constitué.

prototype d'une fonction
: La description de l'interface d'une fonction ; il s'agit du nom de la fonction accompagné de la liste de ses paramètres avec leurs types et de son type de retour.

référencement
: L'action de relever l'adresse d'une donnée afin de la mémoriser dans un pointeur. Le symbole `&` matérialise cette opération.

règle de fabrication
: Une formulation dans un fichier `makefile` qui rassemble une cible, ses prérequis et la commande qui permet de produire la cible. La cible est régénérée par l'exécution de la commande à chaque fois que les prérequis sont modifiés.

structure de données
: Un type librement défini pour représenter l'agglomération de plusieurs variables de types variés. Une structure est généralement associée à un ensemble de fonctions qui définissent le jeu d'opérations usuelles qui ont du sens pour ce nouveau type.

tableau
: Un type pour représenter une séquence de données ayant toutes le même type et qui sont désignées par une unique variable. Un indice permet d'en distinguer les éléments.

tas
: La zone de l'espace d'adressage dans laquelle le système d'exploitation puise pour assurer l'allocation dynamique de mémoire.

transformation lexicale
: L'opération qui consiste à remplacer des portions de codes source par d'autres sans se soucier de la signification de ce texte pour le langage. Le préprocesseur est dédié à de telles transformations.

type opaque
: Un type dont nous ne connaissons que le nom mais pas les détails. Il ne peut être désigné que par des pointeurs qui sont passés à des fonctions spécialisées dans l'usage de ce type (et qui en connaissent les détails). Les flux standards (type `FILE`) sont des types opaques.

unité de compilation
: L'ensemble du code qui est analysé par le compilateur lorsqu'il produit un unique fichier objet. Il s'agit du fichier de code source et de tous les fichiers d'en-tête qu'il inclut.

warning

Un message d'avertissement qui est produit par le compilateur pour signaler une construction qui, sans être strictement interdite, est une source d'erreur ou d'ambiguïté probable. Si un tel message n'empêche pas la compilation ; il faut néanmoins l'analyser avec attention et chercher à l'éliminer.