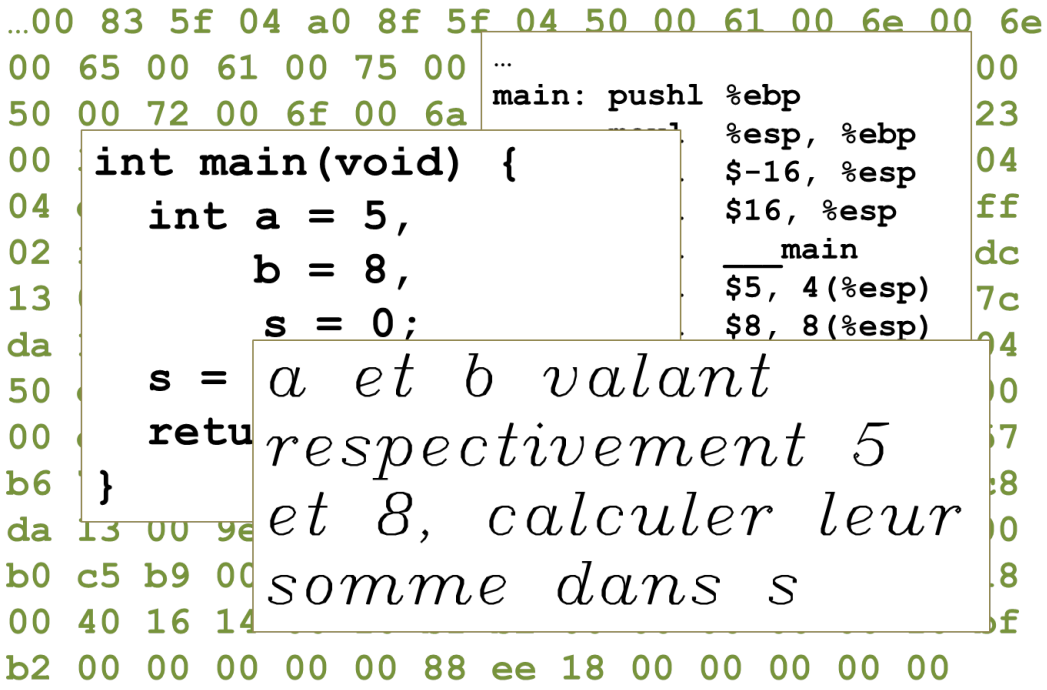
Eléments de cours - Année 1

Introduction à l'algorithmique et à la programmation Application au langage C



1er 2018 - 2019

Ce support n'a pas l'ambition d'être un manuel de référence, mais présente l'essentiel des notions utiles au programmeur débutant. Il présente les notions algorithmiques de base ainsi que leur traduction en langage C, puis les fonction plus spécifiques du C qui s'appuient sur les notions précédentes.

Objectifs

Les compétences ciblées seront les suivantes :

- savoir *analyser* un problème et *concevoir* un algorithme optimal de sa résolution ;
- savoir *traduire* un algorithme en un programme informatique : le langage C sera utilisé ici ;
- savoir *diagnostiquer* les problèmes rencontrés lors de l'exécution d'un programme.

Indications

Différentes boites seront présentées, chacune ayant un rôle informatif différent.

Définition d'un terme
. . . retenir . . .

Conseil
. . . conseil, recommandation, . . .

Attention
. . . prendre garde . . .

Danger
. . . éviter absolument . . .

Représentation des algorithmes

La notation algorithmique n'est pas normalisée, mais la forme suivante est généralement rencontrée :

Algorithme 1 Compter de 1 à 10

```
1: i : entier                                  ▷ i est un entier
2: i ← 1                                       ▷ initialiser i à la valeur 1
3: tant que i <= 10 faire                      ▷ Tant que i est inférieur ou égal à 10
4:     Ecrire i                                ▷ afficher i
5:     i ← (i + 1)                             ▷ incrémenter i
6: fin tant que
```

Langage de programmation et représentation des codes sources

Le langage de programmation utilisé sera le C : les éléments essentiels pour un apprentissage correct du langage seront présentés.

Exemple d'un code source complet (compilable et exécutable) :

Code source 1 – Compter de 1 à 10 en C

```c
/*
 * @but : compter de 1 à 10
 * @auteur : moi
 * @date : 2017-09-01
 *
 */
#include <stdio.h>
/* fonction principale */
int main(void)
{
        int i = 1;
        while (i <= 10) {
                printf("%d",i);
                ++i; // ou : i = (i + 1);
        }

        return 0;
}
```

La commande de compilation des codes sources est la suivante (ici pour le fichier nommé *source* :

1. compilation
2. création du code exécutable

Voir section 20 page 149

Code source 2 – Commande de compilation (gcc)

```
gcc -std=c99 -Wall -Wextra -Wpedantic -c source.c
gcc -o source.exe source.o
```

D'autres options de compilation peuvent être activées pour des besoins spécifiques (cf. documentation `gcc` : https://gcc.gnu.org/onlinedocs/).

Une comparaison des codes sources C avec d'autres langages sera parfois présentée :

Code source 3 – code C

```c
...
int i = 0;
i = 1;
while (i <= 10) {
        printf("␣%d", i);
        i = i + 1;
}
...
```

Code source 4 – code C++

```cpp
...
int i = 0;
i = 1;
while (i <= 10) {
        std::cout << "␣" << i;
        i = i + 1;
}
...
```

Code source 5 – code Python

```python
...
i = 1
while i <= 10 :
        print ("␣", i, end='')
        i = i + 1
...
```

Code source 6 – code VBA

```
...
dim i as integer
i = 1
do while i <= 10
        debug.print "␣",i
        i = i + 1
loop
...
```

Code source 7 – code Cobol

```
...
01 i PIC 9(2).
...
move 1 to i.
perform with test before
                    until (i>10)
        display "␣",i
        add 1 to i
end-perform.
...
```

Les codes sources ont été testés avec les versions suivantes des différents langages de programmation :

- C99 (ISO/IEC 9899 :1999)
- C++11 (ISO/IEC 14882 :2011)

Table des matières

Liste des Algorithmes

Liste des syntaxes

Liste des codes sources

Table des figures

Liste des tableaux

Première partie

Introduction

1 Introduction à l'algorithmique et à la programmation

Un algorithme décrit la succession des actions élémentaires qui, mécaniquement appliquées, permettent d'obtenir un résultat souhaité à partir d'éléments de base. Il est généralement conçu pour être exécuté plusieurs fois. Le résultat doit être identique à chaque exécution, pour les mêmes éléments de base, dans des conditions de fonctionnement identiques et en un temps d'exécution déterminé.

On appelle aussi ce plan d'exécution : procédure, mode opératoire, recette, programme, etc.

1.1 Un exemple : une recette de cuisine

Comment réaliser une succulente tarte au chocolat ? Quels sont les ingrédients nécessaires et quelle est la succession d'étapes à suivre pour parvenir, au bout d'un certain temps, à obtenir le résultat attendu ?

Un grand cuisinier en propose une recette [1] que des milliers de cuisiniers, professionnels ou amateurs, sont maintenant capable de réaliser (voir Figure 1 en page 2)

La préparation, succession des étapes de la réalisation, va utiliser et transformer les ingrédients afin de produire le résultat attendu (voir Figure 2 en page 3).

Cependant, cette recette n'est pas assez précise pour être qualifiée d'algorithme : en effet, des actions comme "laisser cuire doucement" ou bien "vérifier la cuisson en tapotant légèrement" ne sont pas assez précises et rigoureuses pour satisfaire les conditions d'un véritable algorithme qui se doit :

- d'*être non ambigu* : ici "en tapotant légèrement" n'aura pas le même sens pour Hulk et pour Blanche-Neige !
- de *fournir résultat identique à chaque exécution* : je n'arrive pas à obtenir une tarte au chocolat aussi exquise que celle de F. Anton ... c'est donc qu'il manque des informations précieuses qui n'ont pas été communiquées, ou des imprécisions (un tour de main ou des étapes implicites pour lui ...)
- et de fournir ce résultat *dans un temps défini* et acceptable.

Mais nous traiterons, pour notre part, d'*algorithmes informatiques*.

1.2 Algorithme, procédé systématique

1.2.1 Historique

La notion d'algorithme remonte à l'antiquité[2] et décrivait des méthodes de résolution d'équations. Plus récemment (3 siècles av. JC), le mathématicien grec Euclide a décrit la détermina-

[1] source : Recette de la tarte au chocolat, page consultée le 18/08/2016

[2] 3 millénaires avant JC, chez les babyloniens, en Mésopotamie, actuellement l'Irak

FIGURE 1 – Recette de la tarte au chocolat : ingrédients et étapes de réalisation

Ingrédients

- 180 g de farine
- 70 g de sucre glace
- 75 g de beurre
- 3 jaunes d'oeuf

Pour la ganache

- 50 g de lait
- 120 g de crème fleurette
- 120 g de chocolat à 60%
- 12 g de beurre
- 2 petits oeufs

Préparation

Pour la pâte à tarte (vous pouvez également acheter une pâte sablée ou brisée toute faite) :

Tamiser la farine et le sucre glace. Mélanger le beurre pommade (le faire sabler). Quand il ne reste plus de gros morceaux de beurre, incorporer les jaunes d'oeufs. Réserver au froid pendant 30 minutes. Abaisser à 3 mm d'épaisseur et foncer un cercle de 18 cm de diamètre. Cuire la pâte à blanc à 160° pendant environ 10 minutes. La sortir du four et la laisser refroidir.

Pour la ganache :

Préchauffer le four à 170°C.

Concasser le chocolat dans un cul de poule. Dans une casserole, faire bouillir le lait, le beurre et le crème. Verser ce mélange sur le chocolat. Mélanger délicatement en veillant à ne pas faire de bulles. Battre légèrement les oeufs, puis les ajouter à la préparation. Verser ce mélange dans le fond de la tarte préalablement cuite à blanc, éteindre le four et enfourner pendant 16 minutes pour laisser cuire doucement à four éteint.
Fermer le four, laisser cuire 15 minutes.
Vérifier la cuisson en tapotant légèrement, la tarte doit être légèrement tremblotante. Et sinon vous pouvez remettre la tarte en cuisson cinq à quinze minutes de plus.

La recette de la **tarte au chocolat en vidéo**

tion du plus grand commun diviseur de 2 nombres sous forme d'une suite de calculs.

Un exemple du calcul utilisant un algorithme amélioré (l'algorithme d'Euclide)

Algorithme 2 L'algorithme d'Euclide : Plus Grand Commun Diviseur (soustractions)

1: **Fonction** PGCD(a, b) ▷ le PGCD de a et b
Pre-condition: $((a > 0) et (b > 0))$
2: **tant que** $a \neq b$ **faire** ▷ Le calcul est terminé quand $a = b$
3: **si** $a > b$ **alors**
4: $a \leftarrow (a - b)$
5: **sinon**
6: $b \leftarrow (b - a)$
7: **fin si**
8: **fin tant que**
9: **return** a ▷ Le PGCD est la valeur de a
10: **fin Fonction**

FIGURE 2 – Recette de la tarte au chocolat : résultat attendu

On peut tester cet algorithme :

	a	b
valeurs de départ	100	25
	75	25
	50	25
	25	25
pgcd(100,25) =	25	

Algorithme 3 L'algorithme d'Euclide : Plus Grand Commun Diviseur (divisions)

1: **Fonction** PGCD(a, b) ▷ le PGCD de a et b
Pre-condition: $((a > 0) et (b > 0) et (a > b))$
2: r : entier
3: **tant que** $b > 0$ **faire** ▷ Le calcul est terminé quand $b <= 0$
4: $r \leftarrow (a \quad mod \quad b)$
5: $a \leftarrow (b)$
6: $b \leftarrow (r)$
7: **fin tant que**
8: **return** a ▷ Le PGCD est la valeur de a
9: **fin Fonction**

On peut tester cet algorithme :

	a	b	r
valeurs de départ	100	25	
	25	0	0
pgcd(100,25) =	25		

Mais c'est bien plus tard qu'on a pu nommer ces démarches "algorithme". C'est en effet du nom d'un mathématicien perse (800 ap. JC, actuel Ouzbékistan), Al-Khawarizmi, qu'on a introduit, 600 ans plus tard, vers 1550, le terme "algorithme"[3].

[3]Les algorithmes ont été inventés pour transmettre la manière de réaliser un calcul, puis pour automatiser son exécution avec l'apparition des ordinateurs

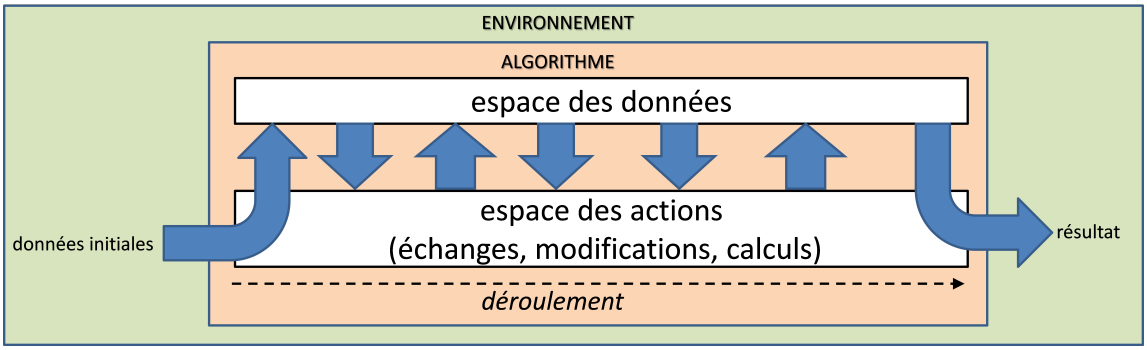
Algorithme

Un algorithme (en anglais : *algorythm*) est un enchaînement ordonné d'actions élémentaires permettant l'obtention d'un résultat déterminé, en un temps fini.

Au même titre que notre recette, un algorithme informatique définit 2 espaces (voir Figure 3 en page 4) :

- l'espace des *éléments de base* (ingrédients), ici les *données* mises en oeuvre dans la résolution d'un problème,
- l'espace des *actions* (préparation) , ici les instructions appliquées aux données, permettant d'opérer des transformations (calculs, etc.) afin d'obtenir le résultat attendu.

FIGURE 3 – Algorithme : espaces des données et des actions



À un problème donné, peuvent correspondre plusieurs algorithmes qui fournissent le bon résultat : il sera nécessaire de choisir le meilleur d'entre eux en étudiant leurs caractéristiques de *complexité* (voir chapitre 17 en page 130)

Algorithmique

L'*algorithmique* est le domaine de l'informatique qui s'intéresse à "l'art, l'ensemble de théories, les règles et techniques permettant de concevoir des algorithmes".

Compléments sur l'architecture des ordinateurs et sur le codage de l'information : voir section 18 page 133.

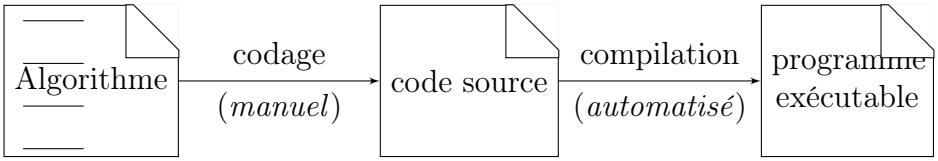
1.3 Algorithme et programme informatique

Un *algorithme informatique* est un enchaînement d'actions élémentaires nécessaire à la résolution de *problèmes* généralement *calculatoires* et dont l'exécution sera confiée à un ordinateur. Pour cela, l'algorithme devra être traduit en langage machine, le seul langage compris par l'ordinateur, et deviendra un programme informatique.

L'algorithme représente la conception, la maquette, le plan, d'un programme informatique à venir.

Dans un algorithme informatique, on s'attachera à lister les instructions à faire exécuter par une machine automatique (= les ordres à lui donner), l'ordinateur, en vue d'obtenir un résultat souhaité.

FIGURE 4 – De l'algorithme au programme exécutable

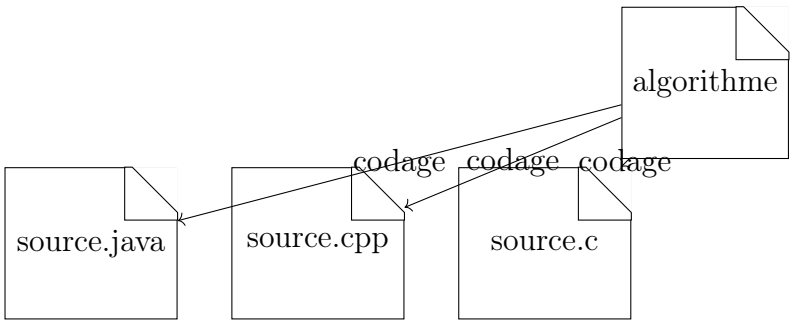


algorithme

Un *algorithme informatique* est le résultat de la conception de la résolution d'un problème calculatoire. Il est exprimé souvent par écrit, sur un support papier, dans la langue maternelle du développeur mais sous une forme compacte qui doit respecter certaines conventions.

Une fois l'algorithme informatique validé, il est traduit dans un langage de programmation et mémorisé dans un fichier texte : c'est le "code source". Ce code source sera ensuite compilé pour devenir un programme exécutable par l'ordinateur (ou tout autre automate programmable)[4]

FIGURE 5 – Code source universel (prog.c) et codes exécutables spécifiques



1.3.1 Programme source

Code source

Un *programme source*, ou *code source*, est le résultat de la traduction d'un algorithme dans un *langage de programmation*. La *programmation* (ou codage) correspond à cette tâche de traduction. Elle est réalisée par un professionnel de l'informatique, le *programmeur*.

Le code source est stocké dans un fichier au format texte brut[5] dont l'extension dépend du langage de programmation utilisé (en C : .c et .h ; en C++ : généralement .cpp ou .cc et .hpp ; en Java : .java ; en Python : .py ; en Cobol : .cob ; etc.).

(cf. section 27 page 157 pour plus d'informations sur les langages de programmation)

[4]Les premiers programmes de l'ère de l'informatique ont été écrits directement en langage machine, mais cette opération était très longue et très fastidieuse et comportait d'importantes risques d'erreurs. C'est pourquoi les langages informatiques ont été introduits, pour être plus proches des langues naturelles

[5]Un fichier texte brut ((en anglais : *anglais : plain text*)) est un fichier dont les caractères sont encodés selon un jeu de caractère connu. Il comporte une suite de caractères affichables (lettres, chiffres, caractères de ponctuation, espace) ; on peut ouvrir ce fichier en utilisant un simple éditeur de texte comme BlocNote. Son contenu ne possède pas de structure à priori : c'est ici la syntaxe du langage de programmation qui va structurer son contenu. Il est utilisé pour tous les codes sources.

1.3.2 Programme exécutable

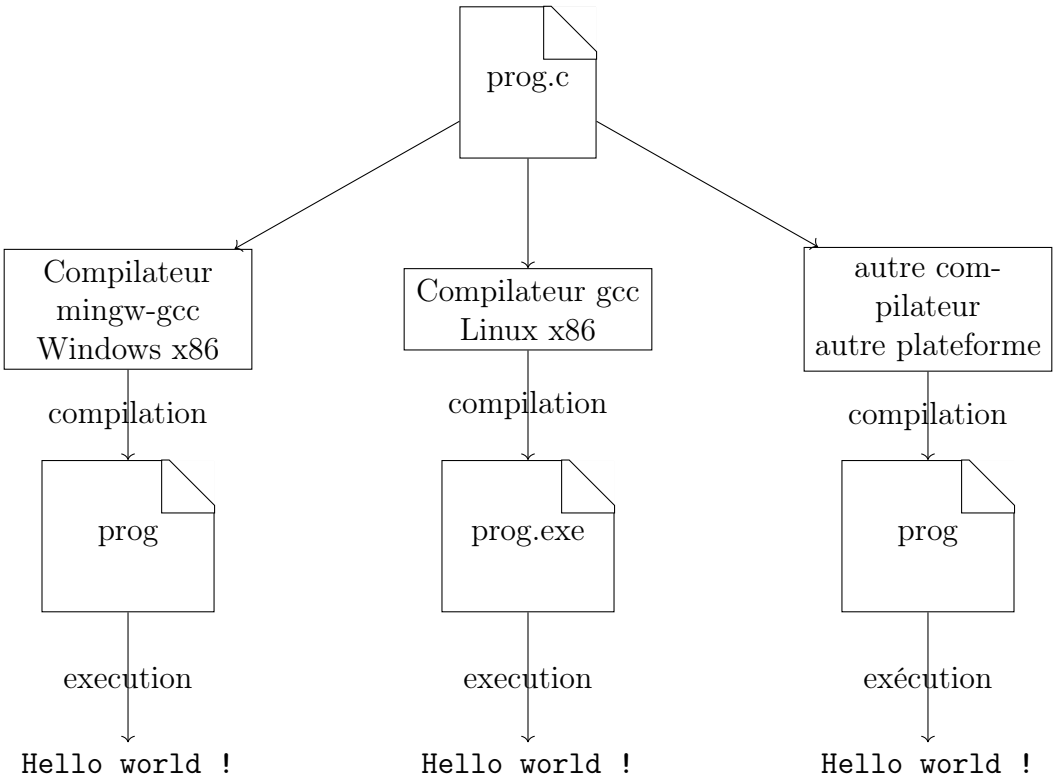
Programme exécutable

Un *programme exécutable* est le résultat de la *traduction d'un programme source* en un ensemble d'instructions exprimées en *langage machine* et directement exécutables par l'ordinateur (ou tout autre automate programmable).

La *compilation* correspond à cette tâche de traduction. Elle est automatisée et est assurée par un ensemble de programmes parmi lesquels le *compilateur*.

Il est stocké dans un fichier binaire exécutable[6], son extension dépend du système d'exploitation de l'ordinateur (.exe sous Windows, pas d'extension sous Linux mais un attribut de fichier).

FIGURE 6 – Code source universel (prog.c) et codes exécutables spécifiques



Logiciel, application informatique

Un *logiciel* (ou application informatique) est un programme exécutable (ou un ensemble de programmes exécutables) utilisé pour répondre aux besoins d'un utilisateur final pour l'aider dans son activité.

1.3.3 Les outils du programmeur

Pour mener à bien l'étage de conception d'un algorithme, on doit disposer d'un simple feuille de papier et d'un crayon.

[6] Un fichier binaire exécutable contient des valeurs qui seront interprétées - comme instructions ou données - par le microprocesseur au moment de l'exécution, après son chargement en mémoire vive de l'ordinateur.

Pour mener à bien l'étape de programmation, on doit disposer au minimum des outils suivants :

1. *un éditeur de texte* : logiciel permettant la création et la modification du code source sous la forme d'un fichier texte (suite de caractères sans mise en forme, non formaté) ; un bon éditeur de texte, pour la programmation, doit offrir
   (a) la *coloration syntaxique*, coloration des mots-clefs du langage de programmation cible,
   (b) la *complétion de code* : proposition de la syntaxe des instructions en fonction du contexte de saisie.

   Par exemple : (a) sous Windows : NotePad++, Geany, Scite, etc. (b) sous Linux : Gedit, Emacs, Vim, etc.
2. un *compilateur du langage C* : (a) sous Windows : MinGw (Minimalist GNU for Windows), (b) sous Linux : gcc (GNU Compiler Collection), associé à un débogueur, logiciel permettant la recherche et la correction des défauts de conception du programme conduisant à des dysfonctionnements ou bogues (en anglais : *bug*)

Dans le cadre de projets informatiques plus complexes, certains outils seront plus adaptés :

- un EDI (Environnement de Développement Intégré (en anglais : *IDE, Integrated Development Environment*) : outil intégrant un éditeur de texte et des liens pour lancer la compilation et utiliser des outils de débogage ; Code : :blocks est un bon EDI sous Windows et sur Linux.
- un AGL (Atelier de Génie Logiciel) (en anglais : *CASE tool, Computer Aided Software Engineering tool*) : il intègre toutes les phases du développement des applications à l'échelle de l'entreprise

1.4 Démarche

Comme toute production, la fabrication d'un produit logiciel (un programme ou un ensemble de programmes) nécessite la mise en oeuvre d'une démarche comportant une succession d'étapes[7] incontournables mais qui seront plus ou moins développées et approfondies selon le niveau de complexité du problème à résoudre. (voir Figure 7 en page 8)

Le problème est généralement défini dans un cahier des charges (cela correspond à l'*expression des besoin*), document qui sert à formaliser les besoins exprimés par un demandeur - le client - afin de s'assurer de la bonne compréhension du problème à résoudre ou à traiter.

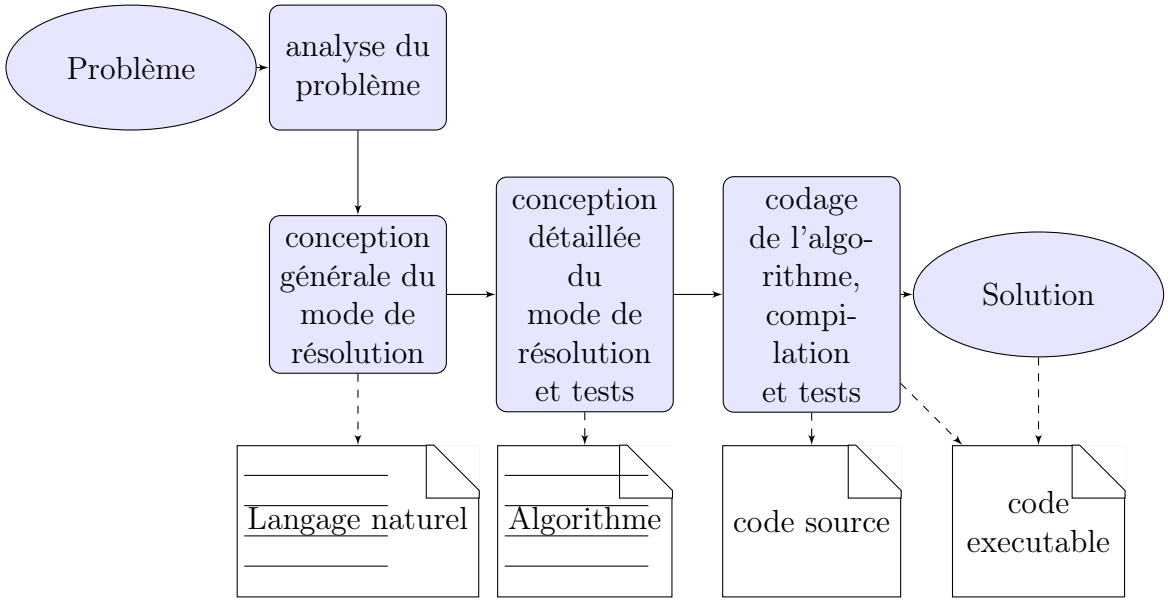
1.4.1 Analyse du problème

Cette étape définit de manière précise, complète, cohérente et non ambiguë le problème posé :

- *analyser* l'énoncé du problème : lire, questionner, comprendre afin de lever toute ambiguïté

[7] c'est un peu comme un algorithme...

FIGURE 7 – Étapes de construction d'un programme à partir de l'expression d'un problème



- *décomposer* un problème complexe en sous-problèmes plus simples (on parle d'*analyse descendante*)
- *reformuler* le problème de manière formelle

A l'issue de cette étape, on obtient une expression du problème claire et non ambiguë, s'appuyant si possible sur une représentation formelle.

1.4.2 Conception générale de la solution

Cette étape recherche les modes de résolution connus du problème et de ses sous-problèmes identifiés précédemment :

- *découvrir* les modes de résolution possibles
- *évaluer* les modes de résolution
- *choisir* un mode de résolution optimal
- *proposer* une organisation globale de la résolution

Ces spécifications décrivent ce que qui devra être réalisé (le *quoi* et non pas *comment*

A l'issue de cette étape, on obtient une forme d'algorithme exprimé dans la langue naturelle : il liste la suite les grandes opérations que l'algorithme devra réaliser (le *quoi*, ce que le programme devra faire).

1.4.3 Conception détaillée de la solution

Cette étape va être plus précise et proposer un algorithme détaillé de la solution retenue :

- *spécifier* les données utiles à la résolution : types des données, noms, valeurs constantes, précisions sur l'utilisation, etc.

- *préciser* l'enchainement optimal des opérations
- *tester* l'algorithme

Création d'une fiche procédure manuelle Les étapes précédentes pourraient aboutir à la création d'une fiche papier à destination d'opérateurs en vue d'une exécution manuelle.

Celle-ci est citée simplement afin de bien comprendre la notion d'algorithme qui part d'une possibilité de résolution manuelle (voir Figure 1.5.4.2 en page 16))

Test de l'algorithme/programme L'algorithme doit être testé, et tant que le résultat souhaité n'est pas atteint, ce dernier doit être corrigé puis à nouveau testé (voir 1.5.4 en page 12)

Jeu d'essai

Un *jeu d'essai* recense les ensembles de données permettant de vérifier le bon fonctionnement d'un algorithme ou programme. Ces ensembles de données forment des *scénarios* d'exécution à vérifier.

Il doit s'attacher à être (le plus) complet (possible) et dépasser les limites des valeurs attendues.

La trace d'exécution d'un algorithme permet la vérification d'un scénario grâce à une simulation manuelle d'exécution : la séquence des instructions est parcourue pour faire évoluer la valeur des données jusqu'à la fourniture du résultat ; ce dernier peut être vérifié en le comparant à un calcul de référence (ou à un calcul manuel) (voir section 1.5.4 en page 12)

TABLE 1 – Trace d'exécution d'un algorithme

Etape de l'algorithme	entrée(s)	liste des données	valeur retournée ou affichage à l'écran
0			
1			
2			
etc.			
fin			résultat obtenu

1.4.4 Codage de l'algorithme

Une fois l'algorithme construit et validé, il est traduit dans un langage de programmation cible :

- *traduire l'algorithme* dans le langage cible : on obtient un code source
- *compiler le code source* : si tout se passe bien, le code exécutable est produit, sinon corriger le code source ;
- *tester le bon fonctionnement programme* exécutable : si tout se passe bien, on passe à l'étape suivante, sinon corriger le code source ou revoir l'algorithme !

Au sujet des tests...

se souvenir de la célèbre citation de Edsger W. Dijkstra[a] :

(en anglais : *Testing shows the presence, not the absence of bugs*) "Tester un programme peut démontrer la présence de bogues, jamais leur absence !"

[a] mathématicien et informaticien néerlandais, 1930-2002, qui a eu un rôle important dans le développement de l'algorithmique de des langages de programmation

C'est un élément supplémentaire pour tester soigneusement les algorithmes !

1.4.5 Livraison du programme

Il s'agit finalement de mettre le programme à la disposition de l'utilisateur :

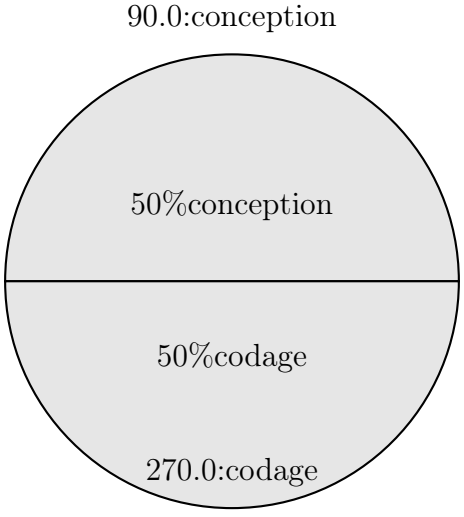
- *installer* le logiciel sur le matériel de l'utilisateur
- *fournir une documentation* "utilisateur" pour expliquer le fonctionnement normal du programme
- *former* les utilisateurs à l'utilisation

Après la livraison, il faudra également prendre en charge

- la *maintenance corrective* du programme en cas d'erreurs de fonctionnement qui n'avaient pas été détectées lors des tests,
- la *maintenance évolutive* afin d'intégrer les améliorations souhaitées.

Le codage de ces évolutions devra vérifier que la correction apporte effectivement une réponse à ce qui était attendu mais aussi qu'elle n'a pas d'effet négatif sur le reste du code.

On peut estimer de manière grossière la répartition des temps accordés à chacune des grandes étapes :



Des études statistiques confirment cette tendance (Cf. étude de Barry Boehm).

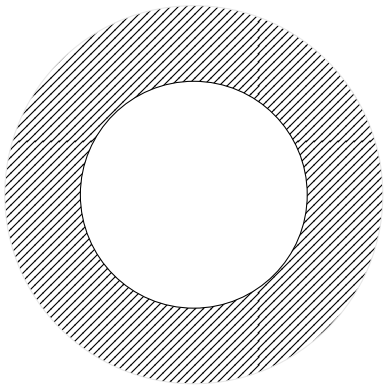
1.5 Exemple

1.5.1 Problème

"Calculer l'aire de la zone de circulation d'un carrefour à sens giratoire" (La zone de circulation correspond à la partie hachurée de la figure 8 en page 11)"

Il est possible de connaitre le diamètre des 2 disques formant cette zone ; nous savons de plus que ces 2 disques sont concentriques.

FIGURE 8 – Aire de la zone de circulation



1.5.2 Analyse du problème

Il s'agit dans un premier temps de reformuler le problème de manière non ambiguë.

Le carrefour peut être représenté par 2 disques circulaires : on peut donc reformuler le problème ainsi, par exemple :

- soient un disque extérieur de diamètre (a), nombre réel, et un disque intérieur de diamètre (b), nombre réel, (b) étant inférieur à (a) ; (c) et (d) étant les aires respectives des disques extérieur et intérieur, il s'agira de calculer $((c) - (d))$
- un sous-problème est évoqué ci-dessus : calculer l'aire d'un disque
- on devra s'assurer que les valeurs saisies pour (a) et (b) soient cohérentes (cette vérification ne sera pas appliquée ici).

1.5.3 Conception générale de la solution

La conception met en évidence 3 éléments :

1. un sous-problème de calcul de l'aire d'une disque ; ce calcul est connu : $(PI \times diametre \times diametre/4)$
2. un sous-problème de calcul de l'aire de la zone de circulation
3. le dialogue avec l'utilisateur pour la saisie des informations et l'affichage du résultat

Algorithme 4 Calculer l'aire d'un disque

Entrée : un réel d, diametre d'un disque
Sortie : l'aire du disque de diamètre d

1: soit le réel constant (PI) ayant pour valeur 3.1415927
2: soit le réel (r) représentant l'aire de ce disque
3: calculer $PI \times d^2/4$ dans (r)
4: l'aire du disque est dans (r)

Algorithme 5 Calculer l'aire de la zone de circulation

Entrée : réels a et b, diamètres des disques extérieur et intérieur
Sortie : l'aire de la piste correspondante

1: (on utilisera le calcul de l'aire d'un disque)
2: soient les réels (c), (d) et (r), respectivement aire du disque extérieur, aire du disque intérieur et aire de la zone
3: calculer l'aire du disque de diamètre (a) dans (c)
4: calculer l'aire du disque de diamètre (b) dans (d)
5: calculer l'aire de la zone de circulation comme différence entre (c) et (d) dans (r)
6: l'aire de la zone de circulation est dans (r)

Algorithme 6 Dialogue principal de calcul de l'aire de la zone

1: soient les réels (diamExt), (diamInt) et (aire), respectivement diamètre du disque extérieur, diamètre du disque intérieur et aire de la zone
2: demander la saisie de (diamExt) et (diamInt)
3: calculer l'aire de la zone de diamètres (diamExt) et (diamInt) dans (aire)
4: afficher l'aire calculée ▷ elle se trouve dans (aire)

1.5.4 Conception détaillée de la solution

On va maintenant détailler les algorithmes précédents de manière plus stricte, en utilisant le formalisme choisi :

Algorithme 7 Calculer l'aire d'un disque

1: **Fonction** CALCULERAIREDISQUE(d : reel) ▷ d, diamètre d'un disque, reel
Pre-condition: $(d > 0)$
2: $PI \leftarrow 3.1415927$
3: r : reel ▷ r : aire du disque, calculé
4: $r \leftarrow 0$ ▷ initialiser r à 0
5: /* calculer l'aire du disque de diamètre (d) dans (r) */
6: $r \leftarrow PI \times d \times d/4$
7: /* retourner l'aire du disque qui est dans (r)*/
8: **return** r
9: **fin Fonction**

Algorithme 8 Calculer l'aire de la zone de circulation d'un carrefour à sens giratoire

1: **Fonction** CALCULERAIREPISTE(a : reel, b : reel) ▷ a, diamètre du grand disque, et b, celui du petit disque
Pre-condition: $((a > b) et (b > 0))$
2: c : reel ▷ c : aire du disque ext., calculé
3: d : reel ▷ d : aire du disque int., calculé
4: r : reel ▷ r : aire de la zone de circulation, calculé
5: /* calculer l'aire du disque de diamètre a dans c */
6: $c \leftarrow calculerAireDisque(a)$
7: /* calculer l'aire du disque de diamètre b dans d */
8: $d \leftarrow calculerAireDisque(b)$
9: /* calculer l'aire de la zone dans r */
10: $r \leftarrow (a - b)$
11: /* retourner l'aire de la zone qui est dans r */
12: **return** r
13: **fin Fonction**

1.5.4.1 Test de l'algorithme/programme : trace d'exécution

La trace d'exécution permet d'effectuer une première validation du bon fonctionnement d'un algorithme (ou d'un programme). Chaque donnée est identifiée, chaque pas de programme est numéroté puis exécuté en modifiant au fur et à mesure les valeurs des variables concernées. Chaque algorithme devra être testé individuellement. L'intégration des algorithmes composant l'ensemble de la résolution devra être testé également.

Algorithme 9 Tester l'algorithme de la fonction 'calculerAireDisque'

1: Ecrire calculerAireDisque(5)
2: Ecrire calculerAireDisque(2)

Algorithme 10 Tester l'algorithme de la fonction 'calculerAirePiste'

1: Ecrire calculerAirePiste(5,2)

TABLE 2 – Trace d'exécution du calcul de l'aire d'un disque de diametre 5

Etape	entrée d		PI			r	valeur retournée
1	5						
	ok						
2			3.1415927				
3						0	
5						19.64	
7							19.64

TABLE 3 – Trace d'exécution du calcul de l'aire d'un disque de diametre 2

Etape	entrée d		PI			r	valeur retournée
1	2						
	ok						
2			3.1415927				
3						0	
5						3.14	
7							3.14

TABLE 4 – Trace d'exécution du calcul de l'aire de la zone de circulation

Etape	entrée a	b		c	d	r	valeur retournée
1	5	2					
	ok	ok					
2				0			
3					0		
4						0	
6				19.64		0	calculerAireDisque(5) (page précédente)
8					3.14	0	calculerAireDisque(2) (page précédente)
10						16.50	
12							16.50
fin							

Algorithme 11 Dialogue principal de calcul de l'aire d'une zone de circulation

1: *diamExt* : reel
2: *diamInt* : reel
3: *aire* : reel
4: Lire *diamExt* ▷ demander la saisie du diametre et mémoriser dans diamExt
5: Lire *diamInt* ▷ demander la saisie du diametre et mémoriser dans diamInt
6: aire ⟵ calculerAirePiste(diamExt, diamInt) ▷ appeler la fonction et mémoriser son résultat dans aire
7: Ecrire "l'aire vaut ", *aire* ▷ afficher le résultat à l'écran

TABLE 5 – Trace d'exécution du calcul de l'aire d'un disque de diametre 2

Etape	Saisies		diamExt	diamInt	aire		valeur affichée
1			0				
2				0			
3					0		
4			5				
5				2			
6						16.5	⟵ calculerAirePiste(5,2)
7							l'aire de la zone de circulation de diamètre extérieur 5 et de diamètre intérieur 2 vaut 16.5

1.5.4.2 Exemple de fiche procédure manuelle

Données demandées

- Diamètre du disque extérieur (a) : ☐ (en mètres), nombre réel
- Diamètre du disque intérieur (b) : ☐ (en mètres), nombre réel

Condition de vérification des données d'entrée

- (a>b) et (b>0)

Données calculées

- Valeur de π (PI) : 3.1415927, nombre réel
- Aire du disque extérieur (c) : ☐ (en m^2), nombre réel
- Aire du disque intérieur (d) : ☐ (en m^2), nombre réel
- Aire de la zone de circulation (r) : ☐ (en m^2), nombre réel

Traitement (suite des opérations et calculs à effectuer)

1. Calculer l'aire du disque de diametre (a) dans (c)
   $c \longleftarrow PI \times a \times a/4$
2. Calculer l'aire du disque de diametre (b) dans (d)
   $d \longleftarrow PI \times b \times b/4$
3. Calculer l'aire de la zone de circulation dans (r)
   $r \longleftarrow (c - d)$

Résultat

- le résultat est la valeur de (r)

1.5.5 Codage de l'algorithme

Code source 8 – Calcul de l'aire d'un disque en C(FnAireDisque.c)

```
1  #include <assert.h>
2
3  /*
4   * @but : fonction de calcul de l'aire d'un disque
5   * @auteur : moi
6   * @date : 2016/09/01
7   * @entree : d , diametre du disque
8   * @précondition : (d>0)
9   * @retour : double , aire du disque
10  */
11 double calculerAireDisque(double d)
12 {
13     /* Précondition */
14         assert(d > 0);
15         /* DECLARATIONS, INITIALISATIONS */
16         const double PI = 3.1415927;
17         double r = 0;
18         /* TRAITEMENT */
19                 /* Calculer l'aire du disque */
20         r = PI * d * d / 4;
21         /* RESULTAT */
22         return r;
23 }
```

où :

- ligne 1 : directive d'inclusion (accès à une bibliothèque externe)
- lignes 3 à 10 : commentaires d'entête de la fonction
- ligne 11 : entête ou prototype de la fonction
- lignes 12 et 23 : jeu d'accolades encadrant le contenu de la fonction, formant le corps de la fonction
- lignes 13 à 22 : corps de la fonction

Conseil

Une *convention de nommage* établit pour un programme, un projet, une équipe oou une entreprise, la manière de nommer les objets d'un programme : les identifiants peuvent être

- clairs ou assez significatifs, comme *diamExt* : ils se suffisent à eux-mêmes,
- simples et non significatifs, comme a : un commentaire devra alors en préciser la

. signification.

Code source 9 – Calcul de l'aire de la zone de circulation en C (FnAirePiste.c)

```
1  #include <assert.h>
2
3  /* Prototypes des fonctions utilisées */
4  double calculerAireDisque(double);
5
6  /*
7   * @but : fonction de calcul de l'aire d'une zone de circulation
8   * @auteur : moi
9   * @date : 2016/09/01
10  * @entree : a , diametre du disque extérieur
11  * @entree : b , diametre du disque intérieur
12  * @précondition : (a>b et b>0)
13  * @retour : double , aire de la zone de circulation
14  */
15 double calculerAirePiste(double a, double b)
16 {
17     /* Précondition */
18         assert (a > b && b > 0);
19         /* DECLARATIONS, INITIALISATIONS */
20         double c = 0,  // aire du disque extérieur
21                d = 0, // aire du disque intérieur
22                r = 0; // aire de la piste
23         /* TRAITEMENT */
24                 /* Calculer l'aire du disque de diametre a dans c */
25         c = calculerAireDisque(a);
26                 /* Calculer l'aire du disque de diametre b dans d */
27         d = calculerAireDisque(b);
28                 /* Calculer la différence entre c et d dans r */
29         r = c - d;
30         /* RESULTAT */
31         return r;
32 }
```

où :

- ligne 1 : directive d'inclusion (accès à une bibliothèque externe)
- ligne 3 : commentaire
- ligne 4 : rappel du prototype de la fonction externe qui sera utilisée dans cette fonction
- lignes 6 à 14 : commentaires d'entête de la fonction
- ligne 15 : entête ou prototype de la fonction
- lignes 16 et 33 : jeu d'accolades encadrant le contenu de la fonction, formant le corps de la fonction

- lignes 17 à 32 : corps de la fonction

Code source 10 – Test de la fcontion calculerAireDisque en C(Test1.c)

```
1  #include <stdio.h>
2
3  /* Prototypes des fonctions utilisées */
4  double calculerAireDisque(double);
5
6  /* fonction principale */
7  /*
8   * @but : tester la fonction de calcul de l'aire d'un disque
9   * @auteur : moi
10  * @date : 2016-09-01
11  *
12  */
13 int main(void)
14 {
15         /* tester la fonction */
16         printf("L'aire de diam. %d vaut %f \n",
17                         3,calculerAireDisque(5)); /* ok */
18         printf("L'aire de diam. %d vaut %f \n",
19                         2,calculerAireDisque(2)); /* ok */
20         printf("L'aire de diam. %d vaut %f \n",
21                         0,calculerAireDisque(0)); /* erreur : (d) n'est pas
22         /* fin de programme de test */
23         return 0;
24 }
```

où :

- ligne 1 : directive d'inclusion (accès à une bibliothèque externe)
- ligne 2 : utilisation d'un espace de nom par défaut
- ligne 4 : commentaire
- ligne 5 : rappel du prototype de la fonction externe qui sera utilisée dans cette fonction
- lignes 7 à 13 : commentaires d'entête de la fonction
- ligne 14 : entête ou prototype de la fonction `main`
- lignes 15 et 22 : jeu d'accolades encadrant le contenu de la fonction, formant le corps de la fonction
- lignes 16 à 21 : corps de la fonction `main`

Code source 11 – Test de la fonction calculerAirePiste en C(Test2.c)

```
1  #include <stdio.h>
2
3  /* Prototypes des fonctions utilisées */
```

```
4  double calculerAirePiste(double, double);
5
6  /* fonction principale */
7  /*
8   * @but : tester la fonction de calcul de l'aire d'une piste
9   * @auteur : moi
10  * @date : 2016-09-01
11  *
12  */
13 int main(void)
14 {
15         /* tester la fonction */
16         printf("Pour %d et %d, l'aire vaut %f \n",
17                         5,2, calculerAirePiste(5, 2)); /* ok */
18         printf("Pour %d et %d, l'aire vaut %f \n",
19                         2,1, calculerAirePiste(2, 1)); /* ok */
20         printf("Pour %d et %d, l'aire vaut %f \n",
21                         1,0, calculerAirePiste(1, 0));
22                         /* ==> erreur : (b) n'est pas >0 */
23         printf("Pour %d et %d, l'aire vaut %f \n",
24                         1,1, calculerAirePiste(1, 1));
25                         /* ==> erreur : (a) n'est pas >(b) */
26         /* fin de programme de test */
27         return 0;
28 }
```

où :

- ligne 1 : directive d'inclusion (accès à une bibliothèque externe)
- ligne 2 : utilisation d'un espace de nom par défaut
- ligne 4 : commentaire
- ligne 5 : rappel du prototype de la fonction externe qui sera utilisée dans cette fonction
- lignes 7 à 13 : commentaires d'entête de la fonction
- ligne 14 : entête ou prototype de la fonction `main`
- lignes 15 et 23 : jeu d'accolades encadrant le contenu de la fonction, formant le corps de la fonction
- lignes 16 à 22 : corps de la fonction `main`

Code source 12 – Dialogue du calcul de l'aire d'une zone C (Appli1.c)

```
1  #include <stdio.h>
2
3  /* Prototypes des fonctions utilisées */
4  double calculerAirePiste(double, double);
5
```

```
6  /* fonction principale */
7  /*
8   * @but : afficher l'aire d'un carrefour giratoire à partir des diametres
9   * @auteur : moi
10  * @date : 2016-09-01
11  * @todo : vérifier la saisie des valeurs des diamètres (cf. préconditions)
12  *
13  */
14 int main(void)
15 {
16         /* DECLARATIONS */
17         int diamExt = 0,
18           diamInt = 0;
19   double aire = 0;
20   /* INITIALISATION */
21     /* demander la saisie contrôlée des diametres */
22         printf("donner le diamètre extérieur :");
23    scanf("%d", &diamExt);
24         printf("donner le diamètre intérieur :");
25    scanf("%d", &diamInt);
26         while (diamExt <= diamInt) {
27                 printf("erreur : le diametre exterieur "
28                     "doit etre supérieur au diametre interieur");
29                 printf("\ndonner le diamètre extérieur :");
30                 scanf("%d", &diamExt);
31                 printf("donner le diamètre intérieur :");
32                 scanf("%d", &diamInt);
33         }
34   /* TRAITEMENT */
35     /* calculer l'aire de la piste */
36   aire = calculerAirePiste(diamExt, diamInt);
37   /* RESULTAT */
38     /* afficher l'aire de la piste */
39   printf( "L'aire de la piste "
40      "de diamètre exterieur %d"
41      "et de diamètre interieur %d "
42      "vaut %f", diamExt, diamInt, aire);
43         /* fin de programme de test */
44         return 0;
45 }
```

où :

- ligne 1 : directive d'inclusion (accès à une bibliothèque externe)
- ligne 2 : instruction indiquant l'utilisation d'un espace de nom par défaut
- ligne 4 : commentaire
- ligne 5 : rappel du prototype de la fonction externe qui sera utilisée dans cette fonction

- lignes 7 à 14 : commentaires d'entête de la fonction
- ligne 15 : entête ou prototype de la fonction principale `main`
- lignes 16 et 38 : jeu d'accolades encadrant le contenu de la fonction, formant le corps de la fonction main
- lignes 17 à 37 : corps de la fonction

Compilation, tests et exécution Lancement de la phase de compilation (vérification de la syntaxe) :

Code source 13 – Commandes de compilation C

```
1 gcc -std=c99 -Wall -Wextra -Wpedantic -c FnAireDisque.c
2 gcc -std=c99 -Wall -Wextra -Wpedantic -c FnAirePiste.c
3 gcc -std=c99 -Wall -Wextra -Wpedantic -c Test2.c
```

Lancement de la phase de production du code exécutable :

Code source 14 – Commandes de création de l'exécutable C

```
1 gcc -std=c99 -Wall -Wextra -Wpedantic
2    FnAireDisque.o FnAirePiste.o Test2.o -o Test2.exe
```

Si la compilation (ou la production de l'exécutable) produit une erreur , il faut corriger les codes sources concernés et recompiler à nouveau.

Sinon, si c'est un avertissement, vérifier sur quelle partie du programme il porte (cela peut cacher une erreur), sinon on peut lancer l'exécution du programme de test :

Code source 15 – Exécution du programme de test

```
1 Test2.exe
```

L'exécution produit le résultat suivant :

```
16.4934
2.35619
Assertion failed: a>b and b>0, ..., line 20
```

On peut ainsi vérifier la précondition. Il faudra cependant compléter le code du programme principal (`main`)qui utilise cette fonction afin de garantir qu'il l'appelle avec les bons arguments.

Il est possible maintenant de compiler l'application qui sera utilisée réellement :

Code source 16 – Commandes de compilation C du dialogue

```
1 gcc -std=c99 -Wall -Wextra -Wpedantic -c Appli1.c
2
3 gcc -std=c99 -Wall -Wextra -Wpedantic
4    FnAireDisque.o FnAirePiste.o Appli1.o -o Appli1.exe
```

puis d'effectuer un test comme utilisateur :

Code source 17 – Lancement de l'application utilisateur (dialogue)

```
1 Appli1.exe
```

Le résultat obtenu est le suivant :

```
donner le diametre exterieur :5
donner le diametre interieur :2
L'aire de la zone  de diametre exterieur 5 et de diametre interieur 2
 vaut 16.4934
```

2 Programme source C

Le langage C définit un certain nombre de composants qui, agencés de manière grammaticalement correcte (règles de la syntaxe), vont constituer un programme source bien formé.

2.1 La langage C

2.1.1 C et C++

Le langage C a été élaboré entre 1969 et 1973 dans les laboratoires Bell par Dennis Ritchie et Ken Thompson pour concevoir le système d'exploitation Unix. Brian Kernighan aida à populariser le langage C en y apportant des modifications. Le système d'exploitation Linux est aujourd'hui principalement basé sur C.

La langage C a subi plusieurs révisions :

- 1989 : ANSI C ou C89, C90 ISO/CEI 9899 :1990 : la version initiale
- 1999 : C99, ISO/CEI 9899 :1999
- 2011 : C11, ISO/CEI 9899 :2011

Chaque version apporte des améliorations et également une tentative de rapprochement avec le langage C++.

Bjarne Stroustrup a crée un nouveau langage, le C++, en 1983, à partir du langage C. C++ est utilisé aujourd'dui dans la conception de nombreux logiciels. C'est un langage qui permet la programmation orientée objets. La langage C a subi plusieurs révisions dont les principales sont :

- 1998 : C++98, ISO/IEC 14882 :1998
- 2003 : C++03, ISO/IEC 14882 :2003 (corrections sur C++98)
- 2011 : C++11, ISO/IEC 14882 :2011
- 2014 : C++14, ISO/IEC 14882 :2014 (améliorations sur sur C++11)

Le C++ reprend la syntaxe de base de C, en grande partie.

2.1.2 Les mots clefs du langage C

Les éléments de grammaire atomiques sont appelés 'token' (ce sont les mots du lexique C).

- spécificateurs de *types* : `char`, `short`, `int`, `long`, `signed`, `unsigned`, `float`, `double`, `void`, enum, `struct`, union, _Bool (alias bool), _Imaginary (alias imaginary), _Complex (alias complex)
- *instructions et structures de contrôle* : `if`, `else`, `switch`, `case`, `default`, `for`, `while`, `do`, `break`, `continue`, `return`, (~~`goto`~~)
- spécificateurs de stockage : auto, register, static, extern
- qualificateurs de type : const, volatile
- autres : sizeof, typedef, `true`, `false`, register, restrict, inline

2.1.3 Les identificateurs

Ils désignent les objets manipulés dans le code source : variables et constantes, fonctions et procédures.

Le choix d'un identificateur est libre, mais parfois lié à une convention de nommage (voir 3.4 en page 30), mais doit, en C, respecter 2 règles :

- il ne doit pas faire partie des mots réservés du langage
- il ne peut comporter que des lettres, des chiffres et le caractère "_", et ne peut commencer par un chiffre.

2.1.4 Les valeurs littérales

représentant les données non identifiées dans un programme (nombres, caractères et chaines de caractères). Elles devraient être évitées dans la mesure du possible : toute donnée devraient en effet être identifiée (mis à part celles auxquelles on ne peut donner de sens : exemple du 0, zéro)

2.1.5 Les délimiteurs

sont des caractères qui ont un rôle de séparateurs dans un code source :

- *point-virgule* ; sépare une instruction de la suivante ou une déclaration d'une autre
- virgule , sépare les éléments d'une liste
- un *couple de parenthèses* () encadre une liste de paramètres
- un *couple de crochets* [] encadre une dimension ou un indice d'un tableau
- un *couple d'accolades* {} encadre un bloc ou une liste de valeurs initiales d'un tableau
- les délimiteurs de commentaires : /∗ et ∗/ encadrent un texte de commentaire ; // définit un commentaire jusqu'au bout de la ligne

- l'antislash \ introduit un caractère spécial ou, en fin de ligne, indique qu'une définition se poursuit sur la ligne suivante
- le point . indique la position décimale dans un nombre, ou sépare un membre d'un de ses composants
- la flèche $->$ donne accès à un membre d'un objet pointé
- :: sélectionne un membre dans un espace de nom (namespace)

Des caractères séparateurs

contribuent à la délimitation de certains mots du langages et à la clarté du programme source : espace, retour à la ligne, tabulation. Ces caractères sont éliminés lors des premières phases de compilation.

Les opérateurs

sont des symboles utilisés pour les calculs

- calculs arithmétiques : +,-,*,/,%
- calculs logiques : ==, !=, >, >=, <, <=, &&, ||
- calculs binaires, décalage binaire : &, |,<<, >>
- dans l'opérateur conditionnel ternaire : ? :

2.2 Structure générale d'un programme C

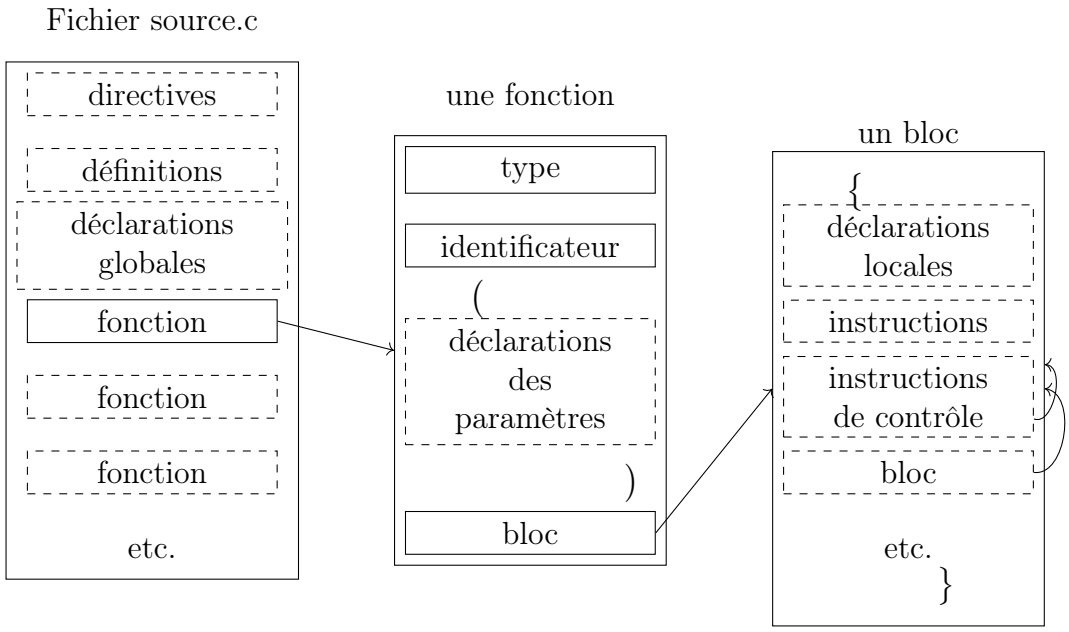
Le code source C d'une application est constitué d'un ou plusieurs fichiers, chacun pouvant comporter les éléments suivants (voir Figure 9 en page 26) :

- des commentaires
- des directives de compilation
- des déclarations (constantes, fonctions)
- des définitions de fonctions, comportant à leur tour :
  - l'entête de fonction
  - le corps de la fonction, composé d'un bloc :
    - délimité par une accolade ouvrante et une accalode fermante
    - et contenant les déclarations de données et instructions associées, et éventuellement d'autres blocs (imbriqués)

2.3 Jeu de caractères et casse des caractères

Le langage C utilise le jeu de caractères US-ASCII : les accents ne seront pas utilisés (sauf dans les chaines de caractères et les commentaires).

FIGURE 9 – Structure générale d'un fichier source C



Le langage C est *sensible à la casse des caractères* : 'Carnaval', 'carnaval', 'carnAval', 'carnavaL' correspondent à différents valeurs. [8] Ainsi 'Main' sera différent de 'main'.

Cette caractéristique s'applique à tous les composants du langage : identifiants des objets, mots-réservés du langage, valeurs littérales, etc.

2.4 Langage typé

C est un langage typé : le type des variables et des fonctions est déterminé à la compilation : cela permet au compilateur de vérifier que les instructions sont cohérentes en regard des types de données manipulés. C intègre les types de données de base : nombres, caractères, valeurs logiques (type booléen introduit en **C99**).

2.5 Les commentaires

Les commentaires sont des explications textuelles ajoutées au programme source afin simplifier de sa maintenance.

Ils indiquent, dans la langue naturelle, ce qui doit être réalisé (le QUOI), le code précisant ensuite la suite d'instructions qui permet la réalisation effective, dans tel ou tel autre langage (le COMMENT).

Choix des commentaires

Les commentaires devrait toujours représenter l'étape de réflexion (algorithme en langue naturelle) qui a précédé l'étape de programmation. Ils participent à la qualité d'un pro-

[8] casse : en imprimerie, boite ou meuble divisé en petites cases contenant des caractères typographiques, 'haut de casse', partie supérieure où sont rangés les caractères les moins utilisés (majuscules/capitales, lettres accentuées), et 'bas de casse', partie inférieure, proche de la main, contenant les caractères les plus courants. D'après Trésors de la langage française

gramme (maintenabilité).

Il ne faut cependant pas que, en trop grand nombre, ils masquent les éléments importants du programme.

Ils devraient également être utiliser pour apporter une explication sur un point d'algorithme complexe.

Code source 18 – Les commentaires sous forme de bloc en C

```
1 /* commentaires bloc,
2    sur plusieurs
3 lignes */
```

Code source 19 – Les commentaires C sous forme de ligne (introduit en **C99**)

```
1 // commentaire sur une ligne
2 .... // commentaire de fin de ligne
```

Conseil

L'utilisation de noms de variables, constantes, fonctions, etc. clairs permet l'élimination de commentaire superflus.

Les commentaires sont totalement ignorées lors de la compilation. Ils ne servent généralement qu'au programmeur. Il existe cependant des logiciels capables d'analyser les commentaires d'un code source pour produire une documentation "claire" d'un programme (Exemple : Doxygen).

2.6 Directives de compilation

Les *directives du préprocesseur* sont des instructions particulières (ne faisant pas partie du langage C lui-même) permettant la réalisation de certaines opérations sur le code source avant la compilation. Elles sont préfixées par le caractère #.

Le pré-processeur est le premier des programmes appelés lors de la compilation. Son rôle est essentiellement :

- d'interpréter les directives de compilation
- de supprimer les commentaires.

Les directives d'inclusion permettent l'insertion de fichiers donnant accès essentiellement à des déclarations de fonctions de bibliothèques C (il s'agit de fichiers à l'extension .h) :

```
1 #include <f>
```

Le nom du fichier (ici f) peut être encadré de chevrons <f> ou de guillemets "f"

- le fichier dont le nom est balisé par des chevrons doit se trouver dans une liste de répertoires système : son chemin (en anglais : *path*) est directement accessible au pré-processeur ;

- le fichier dont le nom est entre guillemets se trouve dans le répertoire courant, ou à une adresse relative (à partir du répertoire courant) ou absolue (à partir de la racine).

Ces fichiers d'entêtes (.h, pour header) contiennent généralement des prototypes de fonctions de bibliothèques C. Le développeur pourra créer ses propres fichiers d'entêtes pour les fonctions qu'il aura développées.

Les macros C permettent de définir des substitutions de valeurs dans le code source avant la phase de compilation :

- substitution simples : toutes les occurrences d'une valeur sont remplacées par une autre valeur : ici PI sera remplacé par 3.1415927

```
1 #define PI 3.1415927
```

D'autres possibilités de substitution existent et sont évoquées en annexe (Cf. annexe 23, page 153)

On s'attachera à limiter l'usage des macros et à utiliser d'autres formes de constantes dans la mesure du possible (Cf. annexe 3.3, page 30).

Deuxième partie

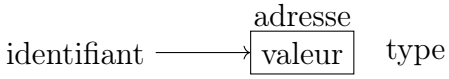
Données

3 Données élémentaires

3.1 Quelques repères...

- *identifiant*, (*id*,*idVar*,*idConst*), ou *identificateur*, ou simplement *nom* : nom unique attribué à un objet (donnée, fonction)
- *valeur* : correspond à un contenu mémorisé (variable ou constante ou valeur littérale) ou calculé (expression)
- *variable* : propriété d'une donnée identifiée dont le contenu est amené à être modifié au cours de l'exécution d'un algorithme
- *constante* : propriété d'une donnée identifiée dont le contenu est fixe tout au long de l'exécution d'un algorithme
- *littéral* ou valeur littérale : valeur non identifiée
- *type*, (T) : nature d'une valeur qui détermine les opérations qui pourront lui être appliquée (= domaine de définition du contenu)

FIGURE 10 – La donnée : son identifiant, sa valeur et son type



Les données représentent, dans un algorithme, les grandeurs et valeurs utiles de la résolution d'un problème.

Une donnée doit être identifiée pour donner accès à sa valeur. De plus l'association d'un type de donnée permet de vérifier que les opérations qui lui sont appliquées sont valides.

A chaque donnée doit correspondre :

- un *identifiant unique* qui donne accès une valeur
- un *type* de donnée précisant la nature de son contenu

3.2 Déclarer des données

Ainsi, comme en mathématiques, toute donnée doit être introduite avant d'être utilisée ("Soit un entier relatif nommé x" : "Soit $x \in \mathbb{Z}$").

3.3 Constante ou variable ?

Il s'agit de déterminer si le contenu d'une donnée est connu à l'avance et ne sera pas modifié au cours de l'exécution de l'algorithme : si c'est le cas, la donnée sera déclarée comme *constante*, dans le cas contraire comme *variable*.

Exemples de valeurs constantes :

- *pi*[9] valant 3.14159...
- *e* valant 2.71828[10]
- tout autre valeur déterminée et invariable au cours de l'exécution (et nécessitant une modification du programme pour être actualisée)

Constante

Une *constante* est une donnée dont la valeur ne changera jamais au cours de l'exécution d'un algorithme.

Exemple de valeurs variables :

- le rayon ou le diamètre lorsqu'on a besoin de calculer des aires de disques, circonférences, etc., sachant que ces données varieront à chaque exécution
- les valeurs de x, a, b et c dans le calcul du discriminant de polynômes du 2nd degré, dont les coefficients seront donnés par l'utilisateur
- toute valeur calculée
- toute valeur saisie par un utilisateur et stockée

Variable

Une *variable* est une donnée dont la valeur est susceptible de changer au cours de l'exécution d'un algorithme.

3.4 Identifiant et conventions de nommage

Le choix de la manière de nommer les objets d'un programmes (constantes, variables, etc.) aura une effet important sur sa maintenabilité.

Convention de nommage

Une *convention de nommage* précise les règles relatives à la manière de nommer des objets (constantes, variables, etc.), convention adoptée par l'ensemble des équipes travaillant dans une entreprise ou autour d'un projet. Ce n'est donc pas une obligation, mais une recommandation.

[9] valeur représentant le rapport constant de la circonférence d'un cercle à son diamètre
[10] constante de Néper, ou nombre d'Euler, base de calcul du logarithme naturel : $ln(e) = 1$

Règles généralement admises pour les identifiant

Un identifiant doit respecter les règles suivantes :

- doit être clair mais concis (cela limite la nécessité des commentaires)
- peut comporter les lettres de 'a' à 'z', de 'A' à 'Z', les chiffres de '0' à '9', le caractère '_' (tiret bas, ou tiret du 8) (il ne doit donc pas comporter d'espaces)
- ne doit pas commencer par un chiffre
- ne doit pas faire partie des mots réservés du langage cible

Une pratique courante (CamelCase) consiste à écrire l'ensemble de mots formant un identifiant de plusieurs lettres en mettant en majuscules la 1ère lettre de chaque mot :

- *lowerCamelCase* : le 1er mot reste en minuscule ; exemple `rayonDuDisque`
- *UpperCamelCase* : le 1er mot commence aussi par une majuscule ; exemple `PointDansLePlan`

Notre convention de nommage

En plus des règles relatives aux identifiant établies plus haut , nous retiendrons la convention :

- *lowerCamelCase* : pour les identifiants des *variables* et des *fonctions*
- *UpperCamelCase* : pour les identifiants des *fichiers* sources, des projets et des *enregistrements*
- EN_LETTRES_CAPITALES : pour les identifiants des *constantes*

3.5 Types de données

Type (de donnée)

Le *type de donnée* détermine la nature du contenu d'une donnée et l'étendue de sa valeur (son domaine de définition).

3.5.1 Types en algorithmique

Les types de données doivent pourvoir représenter toute information à traiter dans la résolution d'un problème : nombres (entiers naturels, entiers relatifs, décimaux, réels), textes, dates, etc.

Cependant, l'algorithmique se contraint généralement aux types qui seront transposables dans un langage de programmation :

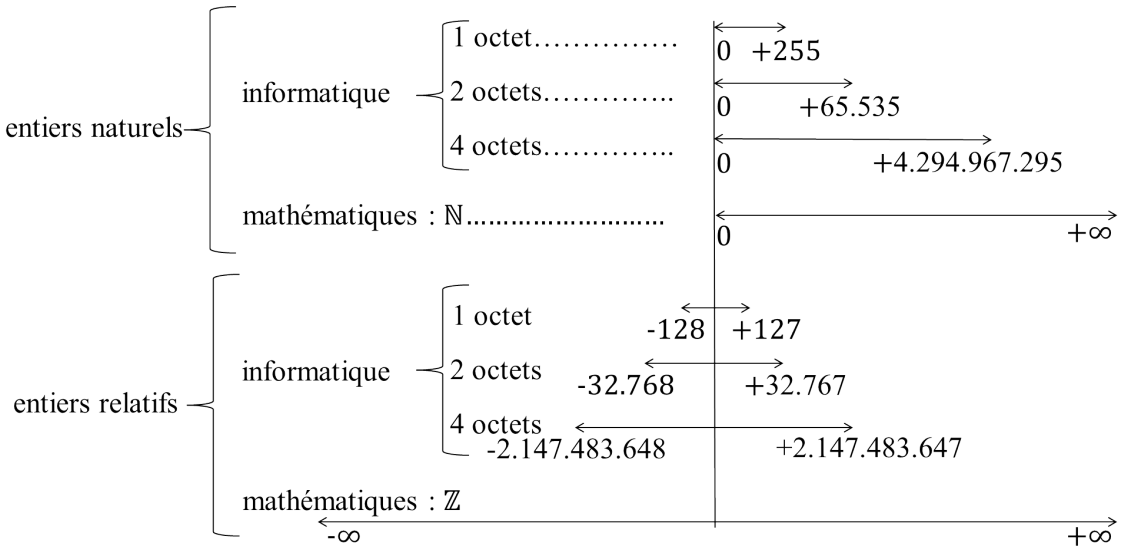
- *entier* : pour les nombres entiers naturels et relatifs
- *réel* : pour tous les autres nombres

- *caractère* : pour représenter un seul caractère
- *chaine* : pour représenter un ensemble de caractères
- *booléen* : pour représenter une valeur booléenne, Vrai ou Faux

Dans certains langages ou certaines versions de langages, ces types ne seront pas disponibles et on devra y substituer un équivalent proposé dans le langage cible (par exemple le type booléen n'existe pas dans le langage C avant **C99** : on doit lui substituer un nombre entier ; le type chaine n'existe pas en C : on doit lui substituer un tableau de caractères).

De plus, l'algorithme ne définit généralement pas de limite à la capacité d'un type en termes de nombres de chiffres et nombre de lettres (ce sont en général les limites mathématiques) : or, le passage aux types des langages introduit une limite de capacité dont il faudra tenir compte. par exemple, voir Figure 11 en page 32.

FIGURE 11 – Domaines de définition et limites de capacité des nombres entiers



3.5.2 Nombres entiers C : `int`

Les nombres entiers permettent de représenter les valeurs de l'ensemble $\mathbb{Z}$ (voir Figure 11 en page 32) limité aux plages de valeurs précisées dans le tableau.

TABLE 6 – Types de base entiers naturels et relatifs en C

	type	Taille (octet)	Plage de valeurs
		signed(par défaut)	
	char	1 o 8 bits	[-128, +127]
	`short` int	2 o 16 bits	[-32768,+32767]
(signed)	`int`	4 o 16-32 bits	[-2147483648, +2147483647]
	`long` int	4 o 16-32 bits	[-2147483648, +2147483647]
(**C99**)			
	`long long` int	8 o 64 bits	[-9223372036854775808 , +9223372036854775807]
		unsigned	
`unsigned`	`char`	1 o	[0, +255]
`unsigned`	`short` int	2 o	[0,+65535]
`unsigned`	`int`	4 o	[0,4294967295]
`unsigned`	`long` int	4 o	[0,4294967295]
(**C99**)			
`unsigned`	`long long` int	8 o	[0,18446744073709551615]

Exemple de valeurs littérales entières :

- représentation décimale (base 10) : -15, 2, 63, 83
- représentation octale (base 8) : 00, 02, 077, 0123 (prefixé par '0' - zéro -)
- représentation hexadécimale : 0x0, 0x2, 0x3f, 0x53 (préfixé par '0x' - zéro x -)

Sur le choix d'un type de donnée

Crash de la mission d'Ariane5 en 1996 : le 4 juin 1996, un débordement d'entier (capacité d'un nombre entier trop petite pour la valeur reçue) provoque l'échec du lancement et l'auto-destruction du lanceur après 37 secondes de vol. L'information de la mesure d'accélération était donnée avec une valeur en 64 bits mais reçue et traitée dans un registre mémoire de 16 bits...

3.5.3 Nombres réels C, virgule flottante : `float`, `double`

Les nombres réels permettent de représenter les nombres de l'ensemble $\mathbb{R}$ dans la limite des valeurs précisées dans le tableau ci-dessous et avec une précision limitée (Cf Annexe 18.4.5.1 en page 145).

Ils sont dits à virgule flottante (en anglais : *floating point*) : la position de la virgule se déplace en fonction de la valeur d'un exposant.

TABLE 7 – Types de base réels en C

type	espace occupé	Plage de valeurs permises
`float`	4 octets	32 bits 1 bit de signe, 8 bits d'exposant, 23 bits de mantisse Simple précision [1.175494e-038, 3.402823e+038]
`double`	8 octets	64 bits 1 bit de signe, 11 bits d'exposant, 52 bits de mantisse Double précision [2.225074e-308, 1.797693e+308]
(**C99**) `long double`	12/16 octets	80/128 bits 1 bit de signe, 15 bits d'exposant, 64/112 bits de mantisse Quadruple précision [3.3621e-4932, 1.18973e+4932]

Exemple de valeurs littérales réelles :

- type `double` par défaut : 1.23, .23, 1., 1.2e-10, -1.8e+15
- suffixe **f** ou **F** pour forcer le type `float` : 2f
- suffixe **l** ou **L** pour forcer le type `long double` : 2l

3.5.4 Nombres décimaux, virgule fixe

Ce type de donnée représente les nombres réels ayant un nombre fini de chiffres à droite de la virgule (ensemble mathématique $\mathbb{D}$, limité à la capacité informatique). Ils sont dits 'à virgule fixe' et sont utilisés essentiellement dans les calculs monétaires dans lesquels la précision est impérative.

Le type décimal à virgule fixe n'est pas défini dans le langage C standard. On utilisera le type réel en sachant que la précision, bien qu'importante, est limitée.

3.5.5 Caractère C : `char`

Ce type de donnée est utilisé pour représenter un caractère du jeu de caractères ASCII, sur une taille de 1 octet (soit 8 bits).

La codification ASCII (voir Figure 22.1 en page 150) définit une table de correspondance entre la valeur numérique d'un octet et le caractère qui lui est associé. Par exemple, une valeur binaire de '0100 0001' (soit 65 en décimal, 41 en hexadécimal) correspond à la lettre 'A'.

L'octet d'une variable caractère contenant une valeur numérique, on peut donc effectuer des opérations arithmétiques sur une valeur de ce type.

TABLE 8 – Type de base caractère en C

type	espace occupé	Plage de valeurs permises
signed char	1 octets	1+7 bits : lettres, chiffres, caractères de ponctuation, etc.
unsigned char	1 octets	8 bits : lettres, chiffres, caractères de ponctuation, etc.

Exemple de valeurs littérales caractères :

- 'a', 'F', '2' , ';'
- int('a') : fournit la valeur numérique du caractère 'a' (voir *transtypage*)

Certains caractères spéciaux (préfixés par \, backslash, antislash) sont utilisés pour définir des valeurs particulières (utilisées essentiellement en C) :

- '\t' : tabulation (espacement horizontal d'environ 4 espaces)
- '\a' : beep
- '\n' : retour à la ligne

Attention

Remarquez l'utilisation du caractère apostrophe (') pour encadrer une valeur littérale de type `char` : 'a', '\n', etc.

3.5.6 Booléens C : `bool` (C99)

Les valeurs booléennes (introduites en **C99**) représentent une donnée logique pouvant contenir l'une des 2 valeurs `true` ou `false`, 1 ou 0 .

TABLE 9 – Type booléen en C

type	espace occupé	valeurs permises
`bool`	1 octets	`true` (1), `false` (0)

`bool` est alias de `_Bool` dans le fichier 'stdbool.h'

Les types numériques entiers peuvent également être utilisés comme des booléens (et vice-versa) pour représenter les valeurs `true` ou `false` :

- toute valeur numérique à 0 équivaut à `false`
- toute autre valeur équivaut à `true`

3.5.7 Chaines (de caractères)

La chaine de caractères peut contenir une suite de caractères, chiffres et autres symboles.

Il n'existe pas de type "chaine de caractères" en C. Cependant, on peut utiliser des valeurs littérales représentées par une suite de caractères, entre guillemets.

Il est également possible de déclarer une variable comme

- pointeur vers le 1er d'un ensemble de caractères :

```
1 char * ch = "Hello world !";
```

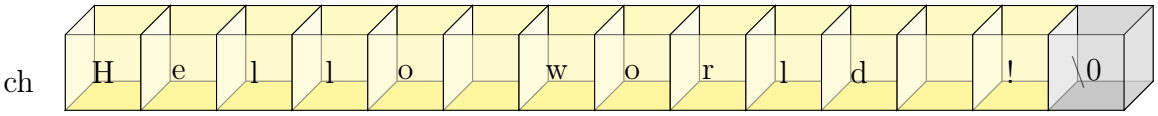
- ou bien comme un tableau de caractères :

```
1 char ch[] = "Hello world !";
```

La variable 'ch' peut ensuite être considérée comme une chaine de caractères en C.

La suite des caractères composant une chaine est encadrée de guillemets :"bonjour", "il est 10h00", "l'informatique est un outil", etc. sont des chaines. Le texte de ce document est une très longue chaine de caractères.

Caractère de terminaison d'un chaine : une chaine C se termine toujours par le caractère nul '\0'. Ce caractère est ajouté pour définir la fin d'une chaine en mémoire. Ainsi, la chaine *ch* précédemment créée peut être représentée ainsi :



3.5.8 Pointeur C

Un pointeur est un type particulier : il définit une variable qui va contenir l'adresse d'une autre variable.

(cf.support Année 2).

3.5.9 Type `void` C

Le type `void` est utilisé essentiellement pour définir le type d'une procédure en C (voir section 15 page 118). Il est également utilisé dans le cas des pointeurs de type non défini (voir section ?? page ??).

`void` en anglais signifie vide, nul.

3.6 Déclaration des variables

Variable

Une *variable* est une donnée identifiée dont le contenu peut être modifié au cours du déroulement d'un algorithme : son type de donnée est fixe, mais son *contenu variable*

On déclare une variable en précisant son type, son identifiant et éventuellement une valeur initiale.

Algorithme 12 Syntaxe algorithmique de déclaration d'une variable

```
1: Variables
2: id : T                                   ▷ id est de type T
```

Par exemple :

Algorithme 13 Exemple de déclaration de variables

```
1: Variables
2: n : entier
3: rayon : reel
4: reponse : caractere
5: msg chaine
6: estNul : booleen
```

Conseil

Il est recommandé de déclarer les variables au début d'une fonction (le C avant le standard **C99**l'impose).

Syntaxe C 3.1 déclaration d'une variable

```
T idVar : T ;
T idVar = expr ;
T idVar1,
    idVar2 = expr,
    idVar3 ;
```

où :

- *T* : un des types de données ;
- *id* : l'identifiant de la variable ;
- *expr* : expression donnant la valeur initiale de la variable ;
- *id1*, *id2* et *id3* : identifiant de 3 variables de même type (seule *id2* est initialisée) ;
- *commentaire* : apporte des précision sur le rôle de cette variable si son identifiant n'est pas significatif.

Initialisation des variables

Le contenu de toute données déclarée mais non initialisée est indéterminé. Il est donc souhaitable d'affecter une valeur initiale au moment de la déclaration : 0 pour un nombre, `true` ou `false` pour un booléen, etc.

Code source 20 – Exemples C de déclaration de variables

```
1 int n = 0;
2 double rayon = 0.0;
3 char reponse = 'o';
4 bool estNul = false;
5 char msg[] = "Hello ! ";
```

3.7 Déclaration des constantes

Constante

Une *constante* est une donnée dont le contenu n'est jamais modifié au cours de l'exécution d'un algorithme. C'est une donnée de référence, *son contenu est fixé et invariable.*

Une constante est une valeur littérale à laquelle un identificateur a été associé.

Conseil

Pour chaque valeur littérale utilisée, demandez-vous si vous pouvez lui associer une signification, un identifiant : si oui, privilégiez la transformation de cette valeur littérale en une constante identifiée.

Algorithme 14 Syntaxe algorithmique de déclaration d'une constante

1: `Constantes`
2: $Tid \leftarrow valeur$

Par exemple :

Algorithme 15 Exemple de déclaration de constantes

1: `Constantes`
2: $N \leftarrow 12$
3: $PI \leftarrow 3.14159$

3.7.1 Directive C `define`

En C, la directive `define` est souvent utilisée pour définir des valeurs constantes. Elle sera privilégiée pour définir la taille des tableaux.

Syntaxe C 3.2 déclaration d'une constante : directive `define`

#define X Y

où :

- `#define` : directive de pré-compilation
- X : mot à remplacer dans le code source du fichier
- Y : valeur par laquelle X est remplacé

Code source 21 – Exemples C de constantes

```
1 #define NB 12
```

Toutes les occurrences de "NB" seront remplacées par la valeur "12", et toutes les occurrence de "PI" seront remplacées par "3.14159" lors de la phase de pré-compilation.

3.7.2 Énumération C : `enum`

L'énumération pourra être utilisée pour définir un groupe de valeurs constantes numériques

Syntaxe C 3.3 déclaration d'une constante : énumération `enum`

```
enum {idConst1, idConst2, etc.};
enum {idConst1 = valeur1, idConst2, etc.};
enum {idConst1 = valeur1, idConst2 = valeur2, etc.};
```

où :

- `enum` : mot réservé pour qualifier une énumération
- *idConst1, idConst2* : identifiants donnés aux valeurs
- *valeur1, valeur2* : valeurs attribuées aux identifiants, par défaut un nombre entier à partir de 0

Code source 22 – Exemples C++ de constantes

```
1 enum {ROUGE, ORANGE, VERT};
2 enum {COEUR, CARREAU, PIQUE=10, TREFLE};
```

donne :

- ROUGE vaut 0, ORANGE vaut 1 et VERT vaut 2.
- COEUR vaut 0, CARREAU vaut 1, PIQUE vaut 10, TREFLE vaut 11

3.7.3 Qualificateur `const`

La qualificateur `const` indique que le contenu d'une variable sera simplement lu et ne pourra être modifié. On parlera en C de variable non mutable.

Syntaxe C 3.4 déclaration d'une constante : qualificateur `const`

```
const T idConst = valeur;
```

où :

- `const` : mot clef pour introduire une constante (plus précisément une *variable non mutable*)
- *T* : type de la constante
- *idConst* : identificateur de la fonctante
- *valeur* : valeur de la constante

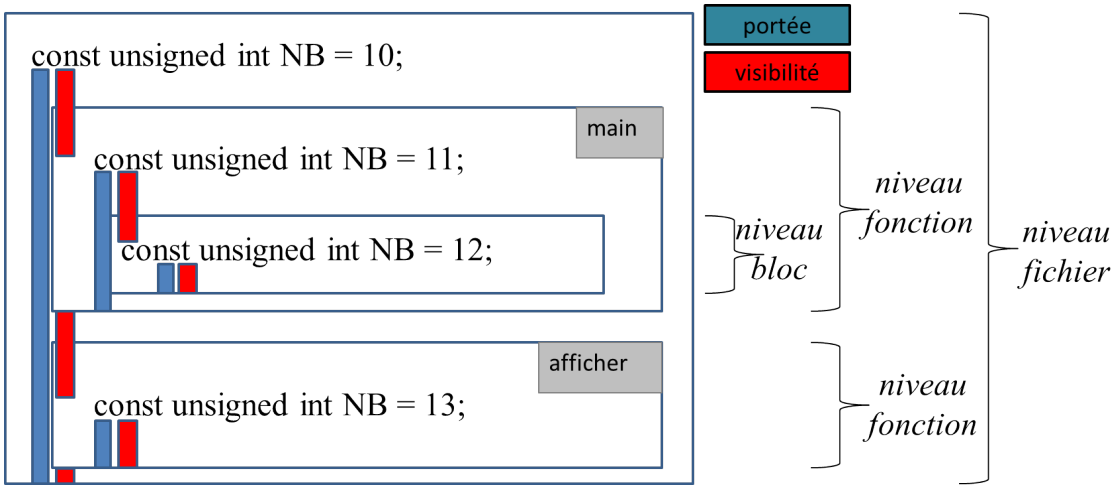
Code source 23 – Exemples C de constantes

```
1 const float PI = 3.1415927;
```

3.8 Portée des déclarations

Les données déclarées ne sont visibles que dans l'algorithme où elles ont été déclarées, et après leur déclaration.

La *portée* d'une déclaration (d'un identifiant) représente son accessibilité à partir du moment où elle a été déclarée. La portée peut être limité par des déclarations dans des blocs imbriquées. (Voir illustration page 40).



Conseil

Les variables seront toujours déclarées à l'intérieur du bloc d'une fonction.

Conseil

Les constantes pourront, si elles sont partagées par plusieurs fonctions, être déclarées globalement, à l'extérieur de toute fonction et en début de fichier source.

3.9 Valeurs littérales

Valeur littérale

Une *valeur littérale* désigne une valeur (nombre, texte, etc.), qui n'est pas nommée : sa signification est soit évidente soit sans importance pour la compréhension du fonctionnement de l'algorithme.

Elle est directement utilisée dans les instructions d'un algorithme.

Attention
Les valeurs littérales seront évitées. Elles seront généralement déclarées comme constantes.

Deux exceptions vont assouplir cette règle :

- des valeurs littérales chaines de caractères pourront être utilisées
- certaines valeurs particulières comme 0 (zéro) : il ne s'agit par de déclarer une constante ZERO valant 0 ; mais, dans certains cas, une constante LIMITE_INFERIEURE valant 0 aurait un sens (une utilité dans la compréhension de l'algorithme).

3.10 Alias (synonyme) d'un type : `typedef`

Un alias (synonyme)sur un type de donnée peut être défini afin de préciser le sens qu'il aura dans un contexte précis.

Syntaxe C 3.5 Déclaration d'un alias sur un type

typedef T alias ;

où :

- `typedef` : mot réservé de redéfinition de type
- *T* : type pour lequel on souhaite ajouter un synonyme
- *alias* : synonyme du type T auquel on peut faire référence dans une déclaration comme type

Par exemple :

```
1 typedef long int grandeDistance;
```

Ce qui nous permettra ensuite d'écrire :

```
1 grandeDistance d1, d2;
```

Cette possibilité est particulièrement utilisée pour renommer les enregistrements (voir section 12 page 104).

3.11 Exercices

1. Parmi les identifiants suivants, indiquer ceux qui sont conformes à la convention, et pour ceux qui ne le sont pas, proposer un identifiant correct

   (a) nom_etudiant
   (b) prenom etudiant
   (c) adresse1
   (d) double
   (e) 2emeAdresse
   (f) moyenne-générale
   (g) moyenne_générale
   (h) long
   (i) moyenneGenerale
   (j) MAX

2. Indiquer le type de donnée de chacune des valeurs littérales suivantes

   (a) 15000
   (b) 5.0
   (c) true
   (d) 'a'
   (e) "nombre entier"
   (f) -5E10
   (g) "false"
   (h) 581236
   (i) 1234.4567
   (j) "entier"
   (k) "12"

3. Effectuer les déclarations des variables permettant de représenter les données suivantes :

   - l'effectif d'une classe d'étudiants
   - la moyenne annuelle d'un étudiant
   - une note est-elle sous la moyenne ou pas ?
   - une mention au Bac
   - la note minimale pour l'obtention du Bac
   - une lettre de l'alphabet

Troisième partie

Opérations élémentaires

4 Opération d'affectation

4.1 Quelques repères...

- *variable* : donnée identifiée dont le contenu peut être modifié
- *expression*, (*expr*) : combinaison cohérente d'opérateurs et d'opérandes qui définissent un calcul, variables, constantes, valeurs littérales, qui produisent une valeur
- *valeur* : résultat de l'évaluation d'un expression

4.2 Affectation

Affectation

L'opération d'affectation remplace le contenu d'une variable avec une nouvelle valeur (du même type).

Algorithme 16 Syntaxe algorithmique de l'opération d'affectation

1: $idVar \leftarrow expr$ ▷ idVar reçoit la valeur de expr

Les 2 parties d'une opération d'affectation sont nommées *left value* et *right value* en regard de la position centrale de l'opérateur d'affectation, à gauche ou à droite :

- left value : c'est la variable réceptrice
- right value : c'est l'expression de la valeur qui sera affectée.

Syntaxe C 4.6 Opération d'affectation

```
idVar = expr ;
```

où :

- *idVar*,*left value* : identifiant de la variable réceptrice de la valeur
- *expr*, *right value* : expression dont la valeur sera copiée dans la variable

Après l'opération d'affectation, la variable réceptrice contient la nouvelle valeur (son ancien contenu est perdu à jamais !).

Remarque : l'opérateur d'affectation s'applique de la droite vers la gauche et renvoie la valeur affectée ; ce qui permet d'écrire :

Syntaxe C 4.7 Opération d'affectations multiples

idVar1 = idVar2 = idVar3 = expr ;

où :

- *idVar1*, *idVar2* et *idVar3* sont les identifiants des variables réceptrices de la valeur
- *expr* : la valeur de l'expression copiée vers la variable cible : successivement idVar3, puis idVar2 et enfin idVar1.

Exemple :

Code source 24 – Exemple C d'affectation

```
1 int i = 0,
2   j = 0; // i et j valent 0
3 i = 1; // i vaut maintenant 1
4 j = 2; // j vaut maintenant 2
5 i = j = 3; // i et j valent 3 (cf. remarque ci-dessous)
```

Attention

L'opérateur d'affectation renvoie la valeur qui a été affectée, d'où la possibilité d'affectations en cascade (ce qui n'est néanmoins *pas conseillé* d'un point de vue lisibilité du code source...)

5 Expressions de calcul

5.1 Quelques repères...

- *expression de calcul* ou *expression* : combinaison cohérente d'opérateurs et d'opérandes qui définissent un calcul.
- *évaluation* : calcul effectif de la valeur d'une expression qui fournit un résultat d'un certain type, en fonction des opérateurs utilisés

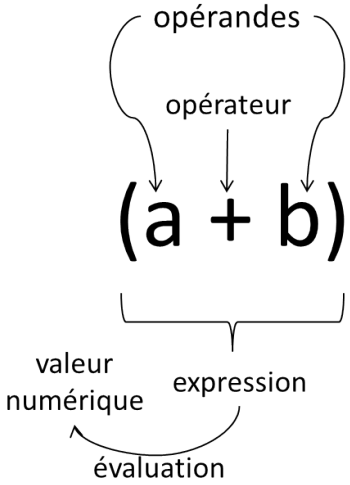
5.2 Expressions numériques, calculs arithmétiques

5.2.1 Calculs arithmétiques en algorithmique

L'algorithmique utilise les opérateurs arithmétique usuels (+,-,*,/) et y ajoute l'opérateur modulo (*mod*) (voir section 5.2.4 page 47) et (*div*) pour la division entière.

Le résultat d'un calcul est évalué pour fournir une valeur (cf. figure 12 page 45), puis peut-être mémorisé dans une variable numérique pour être utilisé plus tard. Si la variable réceptrice est entière alors que le résultat du calcul est un nombre réel, le résultat est tronqué (la partie décimales est perdue).

FIGURE 12 – Expression numérique



Algorithme 17 Syntaxe algorithmique de calculs arithmétiques

```
1: Variables
2: i : entier ← 0
3: Debut
4: i ← 0                          ▷ i vaut maintenant 0
5: i ← (2 + 12)                   ▷ i vaut maintenant 14
6: i ← (i ∗ 2)                    ▷ i vaut maintenant 28
7: i ← ((i + 5) ∗ 2)              ▷ i vaut maintenant 66
8: i ← (i/3)                      ▷ i vaut maintenant 22
9: i ← (i mod 2)                  ▷ i vaut maintenant 0
10: Fin
```

5.2.2 Opérateur unaire

L'opérateur unaire '-' est appliqué à une expression numérique : l'expression résultante a son signe inversé ; l'opérateur unaire '+' n'a pas d'effet sur la valeur mais une expression est quand même calculée.

Syntaxe C 5.8 Opérateur unaire

signe exprNum

où :

- *signe* : le signe à appliquer (seul le signe '-' inverse la valeur)
- *exprNum* : expression numérique à laquelle est appliquée le signe

L'expression résultante

Code source 25 – Opérateur unaire C

```
1 int a = 0,
```

```
2    b = 0;
3  a = 10;
4  b = -a; // b vaut maintenant -10
5  a = -a; // a vaut maintenant -10
6  a = -a; // a vaut maintenant 10
```

5.2.3 Calculs arithmétiques

Ces expressions utilisent des opérateurs arithmétiques et des valeurs numériques élémentaires : leur évaluation renvoie un résultat numérique qui peut être entier, si toutes les opérandes sont entières, ou bien réel si une au moins des opérandes est réelle (Cf. promotion de type : 5.2.7 en page 50).

Syntaxe C 5.9 Expression numérique

(expr1 op expr2)

où :

- *expr1*, *expr2* : opérandes de l'expression numérique
- *op* : l'un des opérateurs arithmétiques

TABLE 10 – Opérateurs arithmétiques C

opérateur	description	exemple	a vaut 5, b vaut 2
+	a plus b	(a + b)	7
-	a moins b	(a - b)	3
*	a multiplié par b	(a * b)	10
/	a divisé par b	(a / b)	2 si a et b sont entiers 2.5 si a, ou b, est réel
%	a modulo b	(a % b)	1 (a et b doivent être entiers)

`a` et `b` doivent être des valeurs numériques. L'opérateur `%` exige des opérandes entières en C

Code source 26 – Exemple d'expression numériques sans affectation en C

```
1  int i = 0;
2  (2 + 12); // valeur calculée 24
3  (i * 2); // valeur calculée 0 (i vaut 0)
4  ((i + 5) * 2); // valeur calculée 10 (i vaut 0)
5  (i / 3); // valeur calculée 0 (i vaut 0)
6  (i % 3); // valeur calculée 0 (i vaut 0)
```

Ces expressions sont calculées, mais leur résultat est perdu ! A moins qu'elles fassent partie d'une expression plus complexe, il est souvent nécessaire de conserver ces résultats en utilisant l'opération d'affectation :

Code source 27 – Exemple d'un calcul arithmétique avec affectation en C

```
1 int i = 0; // i vaut 0
2 i = (2 + 12); // i vaut maintenant 14
3 i = (i * 2); // i vaut maintenant 28
4 i = ((i + 5) * 2); // i vaut maintenant 66
5 i = (i / 3); // i vaut maintenant 22
6 i = (i % 2); // i vaut maintenant 0
```

Dans le cas de calculs utilisant ces opérateurs arithmétiques, et à priorité d'opérateur égale, l'évaluation des calculs intermédiaires a lieu de la gauche vers la droite : ainsi a + b + c calcule d'abord la valeur de a + b, celle-ci est ensuite utilisée dans le calcul suivant.

Danger

En programmation C, il n'y a pas de gestion du débordement de capacité lors des calculs. Avant chaque calcul arithmétique, les données concernées devraient être testées afin de prévoir un éventuel débordement et d'agir en conséquence. (cf. https://www.securecoding.cert.org/con

Les opérateurs de calculs ont des priorités (en anglais : *precedence*) d'application différentes (1 est la plus élevée) :

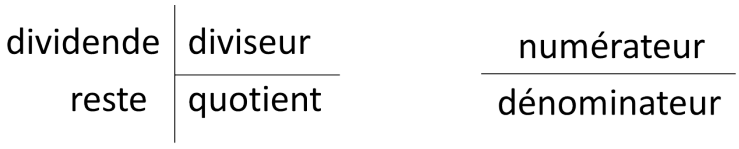
Utilisation des parenthèses

L'usage des parenthèses est fortement conseillé pour définir précisément un calcul surtout en cas d'utilisation d'opérateurs *, / et % associée à des opérateurs de priorité différente comme + et - (voir Tableau des priorités 13 en page 53)

5.2.4 Opérateur modulo

L'opérateur *modulo* calcule le reste de la division euclidienne (ou division entière) de 2 valeurs entières.

FIGURE 13 – Rappel : la division

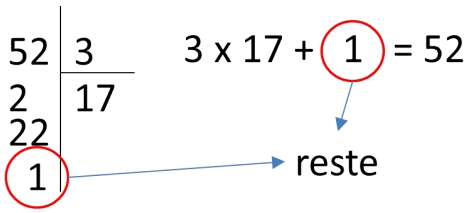


Ainsi 52 (dividende) divisé par 3 (diviseur) donne comme résultat 17 (quotient) et comme reste 1 (reste). Autrement formulé, dividende = diviseur * quotient + reste, soit 52 = 3 * 17 + 1.

Ainsi

- opération de division : 52 / 3 vaut 17 (le quotient)
- opération modulo : 52 % 3 vaut 1 (le reste)

FIGURE 14 – Rappel : le modulo



Code source 28 – Exemple d'un calcul utilisant le modulo en C

```c
int i = 52;
int j = 3;
int q = 0;
int r = 0;

q = (i / j); // q vaut 17
r = (i % j); // r vaut 1
```

L'opérateur modulo est utilisé essentiellement pour tester la divisibilité d'un nombre par un autre.

5.2.5 Affectation composée en C

L'opération de l'affectation sert à conserver le résultat de l'évaluation d'une expression pour une utilisation ultérieure dans le déroulement d'un programme. Certains opérateurs de calcul réalisent simultanément une opération d'affectation.

Les opérateurs d'affectation composée combine un opérateur arithmétique et une affectation :

TABLE 11 – Opérateurs d'affectation composée C

opérateur	exemple	équivalent à
+=	a += b ;	a = (a + b) ;
-=	a -= b ;	a = (a - b) ;
*=	a *= b ;	a = (a * b) ;
/=	a /= b ;	a = (a / b) ;
%=	a %= b ;	a = (a % b) ;

Code source 29 – Exemple d'utilisation de l'affectation composée en C

```c
int i = 1;
int j = 2;

i += 1; /* i vaut maintenant 2*/
i += j; /* i vaut maintenant 4*/
i *= 3; /* i vaut maintenant 12*/
```

La priorité des opérateurs d'affectation composée est identique à celle de l'affectation : elle est moins prioritaires que tous les opérateurs de calculs (arithmétiques et logiques).

5.2.6 Incrémentation et décrémentation en C

Incrémenter

Incrémenter une variable consiste à lui ajouter 1.

Décrémenter

Décrémenter une variable consiste à lui soustraire 1.

Attention

Cette expression de calcul renvoie une valeur qui sera différente en fonction du placement de l'opérateur, devant la variable ou derrière la variable

- préincrémentation/prédécrémentation : l'opération est appliquée à la variable et *sa nouvelle valeur est renvoyée*,
- postincrémentation/postdécrémentation : l'opération est appliquée à la variable mais c'est *sa valeur initiale* qui *est renvoyée*.

TABLE 12 – Préincrémentation et postincrémentation C

opérateur	exemple	équivalent à
$++$	$a++$	ajoute 1 à a et renvoie la valeur initiale de a
	$++a$	ajoute 1 à a et renvoie la nouvelle valeur de a
$--$	$a--$	soustrait 1 de a et renvoie la valeur initiale de a
	$--a$	soustrait 1 de a et renvoie la nouvelle valeur de a

Exemple :

Code source 30 – Incrémentation C

```
1 int i = 1;
2 ++i; // i vaut maintenant 2
3 i++; // i vaut maintenant 3
```

Exemple :

Code source 31 – Décrémentation C

```
1 int i = 3;
2 --i; // i vaut maintenant 2
3 i--; // i vaut maintenant 1
```

Attention

Évitez l'utilisation de ces opérateurs dans des expressions calculées plus complexes, ou dans des expressions logiques, car cela peut amener de la confusion lors de la lecture ou de la maintenance.

Ainsi les 2 exemples suivants sont-ils à éviter :

Exemple :

Code source 32 – Pré- opérations à éviter en C

```
1 int i = 1;
2 cout << ++i; // i vaut maintenant 2, 2 est affiché
3 cout << --i; // i vaut maintenant 1, 1 est affiché
```

Exemple :

Code source 33 – Post- opérations à éviter en C

```
1 int i = 1;
2 cout << i++; // 1 est affiché, i vaut maintenant 2
3 cout << i--; // 2 est affiché, i vaut maintenant 1
```

Conseil

Les opérateurs d'incrémentation/décrémentation devraient être utilisés seuls.

Exemple :

Code source 34 – Pré- opérations à privilégier en C

```
1 int i = 1;
2 ++i; // i vaut maintenant 2
3 cout << i; // 2 est affiché
4 --i; // i vaut maintenant 1
5 cout << i; // 1 est affiché
```

Exemple :

Code source 35 – Post- opérations à privilégier en C

```
1 int i = 1;
2 cout << i; // 1 est affiché
3 i++; //i vaut maintenant 2
4 cout << i; // 2 est affiché
5 i--; // i vaut maintenant 1
```

5.2.7 Transtypage et promotion de type en C

Les opérations de calculs arithmétiques fournissent un résultat du type de la donnée ayant la plus grande capacité.

Code source 36 – Exemple d'un calcul utilisant le modulo en C

```
1 int i = 52;
2 int j = 3;
3 float q = 25;
4 double r = 4;
5
```

```
6 (i / j); // expression de type int
7 (i / q); // expression de type float
8 (i / r); // expression de type double
9 (q / r); // expression de type double
```

Si on souhaite contrarier ce type, il est possible, au sein d'un calcul, de forcer le type d'une valeur qui est utilisée.

Transtypage

Le *transtypage* (en anglais : *cast*) est une opération temporaire de changement du type d'une valeur. Il s'agit de forcer, contrarier, le type d'une valeur du calcul pour modifier le type ou la précision du résultat d'une expression.

Promotion de type

Il y a une *promotion de type* lorsque le transtypage s'effectue "naturellement" vers un type de plus grande capacité.

Code source 37 – Transtypage C

```
1  #include <stdio.h>
2  #include <assert.h>
3
4  /**
5  * @but : fonction de calcul de la moyenne de 2 nombres entiers
6  * @auteur : moi
7  * @entree : a , 1er nombre
8  * @entree : b , 2eme nombre
9  * @retourne : double , moyenne
10 */
11 double calculerMoyenne (int a, int b)
12 {
13         double moyenne = 0;
14         moyenne = (a+b)/2; /* (int+int)/int -> int */
15         return moyenne;
16 }
17
18 /**
19 * @but : fonction de test de la fonction calculMoyenne
20 * @auteur : moi
21 * @retourne : int , status de l'exécution
22 */
23 int main(void)
24 {
25         printf("Moy. de %d et %d est %f \n",
26         10,12,calculerMoyenne(10,12));
27         printf("Moy. de %d et %d est %f \n",
28         10,11,calculerMoyenne(10,11));
```

```
29
30          return 0;
31  }
```

L'exécution produit le résultat suivant :

```
11
10   <-- devrait être 10.5
```

Il n'y a pas de promotion de type, les 2 opérandes sont entières, le résultat est entier. Pour obtenir un résultat de type réel, il faut introduire un transtypage dans l'expression de calcul.

Code source 38 – Transtypage C

```
1   #include <stdio.h>
2   #include <assert.h>
3
4   /**
5   * @but : fonction de calcul de la moyenne de 2 nombres entiers
6   * @auteur : moi
7   * @entree : a , 1er nombre
8   * @entree : b , 2eme nombre
9   * @retourne : double , la moyenne de a et b
10  */
11  double calculerMoyenne (int a, int b)
12  {
13          double moyenne = 0;
14          moyenne = (a+b)/2.0; // (int+int)/double -> double
15          // ou bien :
16          // moyenne = (double)(a+b)/2; // double(int+int)/int -> double
17          return moyenne;
18  }
19
20  /**
21  * @but : fonction de test de la fonction calculMoyenne
22  * @auteur : moi
23  * @retourne : int , status de l'exécution
24  */
25  int main(void)
26  {
27          printf("Moy. de %d et %d vaut %f \n",
28          10,12,calculerMoyenne(10,12));
29          printf("Moy. de %d et %d vaut %f \n",
30          10,11,calculerMoyenne(10,11));
31
32          return 0;
33  }
```

La fonction a été modifiée :

- un élément réel a été ajouté dans le calcul : 2.0 ; la promotion de type va s'appliquer

- un transtypage aurait pu être ajouté pour transformer le résultat de $(a+b)$, qui devrait être entier, en nombre réel (`double`).

```
11
10.5
```

5.2.8 Priorité des opérateurs

Certains opérateurs de calculs s'appliquent en priorité par rapport à d'autres. Il n'est pas simple de retenir l'ensemble des priorités, d'autant plus que certains opérateurs sont évalués de la droite vers la gauche et d'autres de la gauche vers la droite dans une expression.

L'usage des parenthèses dans les calculs permet généralement de faire abstraction de la priorité des opérateurs (en anglais : *operator precedence*).

TABLE 13 – Priorité des opérateurs arithmétiques C

Priorité	Opérateurs
1	$x++$, $x--$
2	$++x$, $--x$, $-x$, $+x$
3	$a*b$, a/b, $a\%b$
4	$a+b$, $a-b$
...	...
14	$=$, $a+=b$, $a-=b$, $a*=b$, $a/=b$, $a\%=b$

Source : http://en.cppreference.com/w/c/language/operator_precedence

Utilisation des parenthèses

L'utilisation des parenthèses dans les expressions numériques lève tout risque d'ambiguïté dans les calculs effectués.

Exemples : pour a et b, entiers, valant respectivement 2 et 5

- $(2*a+b)$: vaut 9 (calcul de (2 * a), puis ajout de b)
- $(2*(a+b))$: vaut 14

5.2.9 Rappel : éléments neutres

Les 2 valeurs suivantes sont intéressantes dans certains calculs : ce sont des éléments neutres, qui n'ont aucune incidence sur le résultat :

- le 0 (zéro) pour l'addition ;
- le 1 (un) pour la multiplication.

Ils servent à initialiser les valeurs des cumuls de sommes ou de produits.

5.2.10 Exercices

1. Utiliser une trace d'exécution pour déterminer la valeur finale des variables x et y à la fin de l'exécution de ces 3 groupes d'instructions(x et y sont des nombres entiers).

1. x = 10 2. y = 5 3. x = y 4. y = x 5. x = x + 1 6. x = -y	1. x = 2 2. y = 4 + x * 2 3. x = x * y - 2 4. y = x - y 5. y = y - 1 6. x = x - y	1. x = 5 2. x = x * 2 3. x = 2 * x + 5 4. y = x - y 5. x = y mod 2 6. x = x + y
x vaut :	x vaut :	x vaut :
y vaut	y vaut	y vaut

5.3 Expressions logiques, tests, conditions

Expression logique

Une *expression logique*, ou *booléenne*, est un calcul dont le *résultat* est une *valeur booléenne* (Vrai/Faux, 1/0, `true`/`false`).

Attention

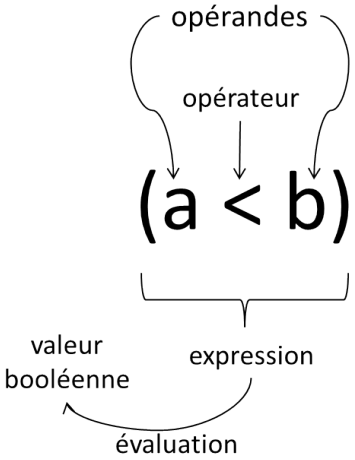
En C, toute valeur numérique entière différente de 0 sera considérée comme vraie (`true`) et à 0 comme fausse (`false`).

Un nombre peut donc être utilisé comme valeur booléenne.

5.3.1 Opérateurs de comparaison

opérateurs de comparaison

Les *opérateurs de comparaison* mettent en relation 2 valeurs numériques élémentaires pour les comparer et retournent une valeur booléenne.



Syntaxe C 5.10 Expression logique

(expr1 opLog expr2)

où :

- *expr1*, *expr2* : opérandes de l'expression logique, qui doivent être compatibles avec l'opérateur de comparaison utilisé
- *opLog* : opérateur de comparaison

TABLE 14 – Opérateurs de comparaison C

opérateur	**description**	**exemple**	si a vaut 2 et b 3
==	a est-il égal à b ?	(a == b)	false
!=	a est-il différent de b ?	(a != b)	true
<	a est-il plus petit que b ?	(a < b)	true
<=	a est-il plus petit ou égal à b ?	(a <= b)	true
>	a est-il plus grand que b ?	(a > b)	false
>=	a est-il plus grand ou égal à b ?	(a >= b)	false

Code source 39 – Exemple d'expression logique

```
1 int i = 2;
2 int j = 5;
3 bool test = false; // test vaut false/0
4
5 test = (i < j); // test vaut maintenant true/1
6 test = (i == j); // test vaut maintenant false/0
7 test = (i > j); // test vaut maintenant false/0
```

Attention

Ne pas utiliser le caractère associatif des opérateurs de comparaison (une comparaison renvoie en effet `true` ou `false`, soit 1 ou 0, ce qui fausse les autres comparaisons)

Ainsi l'exemple suivant est-il à rejeter :

Code source 40 – Expression logique erronée

```
1 int a = 2;
2 int b = 2;
3 int c = 2;
4 bool test;
5
6 test = ( a < b < c ) ;  // test vaut true, false était supposé
7 test = ( a == b == c ); // test vaut false, true était supposé
```

Dans une expression logique, les comparaisons s'appliquent de gauche à droite :

- dans le 1er cas : $(a < b)$ vaut (`false`), soit 0 , puis $(0 < c)$ vaut `true`
- dans le 2ème cas : $(a == b)$ vaut (`true`), soit 1 , puis $(1 == c)$ vaut `false`

Il sera remplacé par :

Code source 41 – Expression logique correcte

```
1 int a = 2;
2 int b = 2;
3 int c = 2;
4 bool test;
5
6 test = ( (a < b) && (b < c ) ) ; // false : OK
7 test = ( (a == b) && (b == c ) ) ; // true : OK
```

Dans le cas de calculs utilisant ces opérateurs de comparaison et à priorité d'opérateur égale, l'évaluation des comparaisons intermédiaires a lieu de la gauche vers la droite. Ainsi a < b < c évalue d'abord a < b qui produit un résultat booléen (mais en fait numérique, 0 ou 1), puis ce résultat est comparé à c.

Il en est de même pour les connecteurs logiques.

Attention

Les comparaisons utilisant des valeurs réelles ne sont pas garanties à cause de la valeur approchée de ces valeurs. (Cf. Annexe 18.4.5.1 en page 145)

Code source 42 – Comparaison erronée entre nombres réels

```
14   double a = 10.1;
15   double b = 10.2;
16   double c = 20.3;
17   /* INITIALISATION / TRAITEMENT */
18   bool test = ((a+b) == c);
19   if (test) {
20     printf("%2.16f + %2.16f egale %2.16f\n", a, b, c);
21   } else {
22     printf("%2.16f + %2.16f diffère de %2.16f\n", a, b, c);
23   }
24   if ((a+b) == c) {
25     printf("%2.16f + %2.16f égale %2.16f\n", a, b, c);
26   } else {
27     printf("%2.16f + %2.16f diffère de %2.16f\n", a, b, c);
28   }
29   if ((10.1 + 10.2) == 20.3) {
30     printf("%2.16f + %2.16f égale %2.16f\n", a, b, c);
31   } else {
32     printf("%2.16f + %2.16f diffère de %2.16f\n", a, b, c);
33   }
```

Résultat :

```
10.1000000000000000 + 10.1999999999999990 diffÞre de 20.3000000000000010
10.1000000000000000 + 10.1999999999999990 diffÞre de 20.3000000000000010
10.1000000000000000 + 10.1999999999999990 diffÞre de 20.3000000000000010
```

Danger

Ne pas confondre l'instruction d'affectation '=' et l'opérateur de test d'égalité '=='.

Le compilateur (à condition qu'il soit correctement paramétré) signale les expressions logiques comportant des affectations par un avertissement :

```
warning: suggest parentheses around assignment used as truth value [-Wparentheses]|
```

Ensemble : un ensemble peut être défini par une liste de valeurs

Par exemple, $3, 8, 12$ est un ensemble de valeurs.

Pour tester qu'une valeur v appartienne à cet ensemble, on utilisera l'opérateur d'égalité associé au connecteur logique OU :

v = 3 ou v = 8 ou v = 12

Ambigüités : dans les expressions "véhicules jusqu'à 2m de hauteur, bagage jusqu'à 23 kg, etc.", est-ce qu'un véhicule de 2m ou un bagage de 23kg peuvent être acceptés ?

Les opérateurs d'inégalité (<, <=, >, >=) sont utilisés pour éclaircir ces expressions ; soit h ; la hauteur d'un véhicule et b, le poids d'un bagage :

- h < 2, 2 est exclu : jusqu'à 2m, mais les véhicules de 2m exactement sont refusés
- h <= 2, 2 et inclus : jusqu'à 2m, les véhicules de 2m exactement sont acceptés
- b < 23, 23 est exclu : jusqu'à 23kg, mais les bagages de 23kg exactement sont refusés
- b <= 23, 23 est inclus : jusqu'à 23kg, les bagages de 23kg exactement sont acceptés

Voir plus bas : les intervalles.

5.3.2 Connecteurs logiques

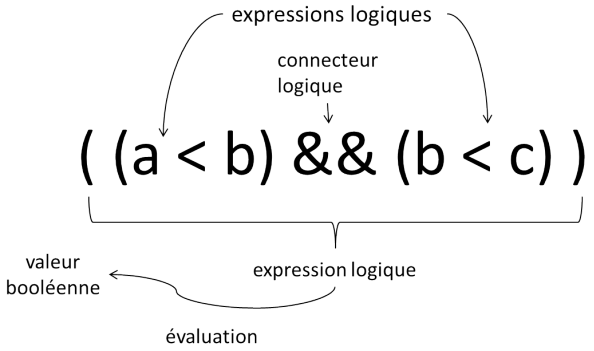
Les connecteurs logiques permettent la construction d'expression logiques plus complexes, en associant plusieurs expressions booléennes. Leur évaluation produit un résultat booléen.

Ainsi, A et B étant des expression logique (valeurs booléennes) :

TABLE 15 – Table de vérité

A	B	A et B	A ou B	non A
vrai	vrai	vrai	vrai	faux
vrai	faux	faux	vrai	faux
faux	vrai	faux	vrai	vrai
faux	faux	faux	faux	vrai

FIGURE 15 – Connecteurs logiques dans une expression



Syntaxe C 5.11 Expression logique : connecteurs logiques

exprLog1 conn exprLog2

où :

- *exprLog1*,*exprLog2* : expressions logiques
- *conn* : connecteur logique

TABLE 16 – Connecteurs logiques C

opérateur	description	exemple
&&	condition1 ET condition2 ?	((a == b) && (b == c)) ((a == b) and (b == c))
||	condition1 OU condition2 ?	((a == b) || (b == c)) ((a == b) or (b == c))
!	NON condition1 ?	!(a < b) not(a < b)

Code source 43 – Exemple d'utilisation des connecteurs logiques en C

```
1  int age;
2  int note;
3  bool test;
4  ...
5  test = ((age >= 18) && (note > 10));
6       // true si age >= 18 ET note > 10, false sinon
7
8  test = ((age >= 18) || (note > 10));
9       // true si age >= 18 OU note > 10, false sinon
10
11 test = (!(age >= 18) || (note > 10));
12      // true si age < 18 OU note > 10, false sinon
```

Intervalles : ensemble des valeurs qui se trouvent entre les 2 bornes d'un intervalle.

Les symboles [et] indique si les bornes sont incluses ou pas ; ainsi pour définir l'ensemble de valeurs x entre a et b :

- $[a, b]$: a et b sont incluses, soit $a <= x <= b$
- $[a, b[$: a est incluse et b est exclue, soit $a <= x < b$
- $]a, b]$: a est exclue et b est incluse, soit $a < x <= b$
- $]a, b[$: a et b sont exclues, soit $a < x < b$

La traduction de ces intervalles utilise les opérateurs d'inégalité combinés avec l'opérateur ET :

- $[a, b] \rightarrow (a <= x) et (x <= b)$
- $[a, b[\rightarrow (a <= x) et (x < b)$
- $]a, b] \rightarrow (a < x) et (x <= b)$
- $]a, b[\rightarrow (a < x) et (x < b)$

La traduction de bornes infinies simplifie l'écriture :

- $[a, \inf[\rightarrow (a <= x)$
- $]a, \inf[\rightarrow (a < x)$
- $]-\inf, a] \rightarrow (a <= x)$
- $]-\inf, a[\rightarrow (a < x)$

L'union de 2 (ou plus) intervalles va définir qu'une valeur appartienne à l'un des 2 (ou plus) intervalles :

exemple : inférieur à 2 ou supérieur à 3

L'intersection de 2 (ou plus) intervalles définit qu'une valeur appartient à chacun des 2 (ou plus) intervalles :

exemple : supérieur à 2 et inférieur à 3

Ordre des expressions logiques : les conditions sont évaluées de la gauche vers la droite ; aussi un agencement cohérent des expressions élémentaires dans une expression plus complexe est important :

- dans le cas `&&` : si la première des conditions n'est pas remplie, les suivantes ne sont pas évaluées (c'est une optimisation)
- dans le cas `||` : si la première des conditions est vérifiée, les suivantes ne sont pas évaluées (c'est aussi une optimisation)

Par exemple, i et j sont des entiers, i ayant une valeur quelconque et j valant 0 :

- dans l'expression : `((j != 0) && ( (i/j) != 0))`

  - `(j != 0)` est évalué d'abord : j valant 0, cette expression vaut `false` ;
  - le connecteur avec la suivante étant un ET logique, il n'est pas nécessaire d'effectuer les tests logiques suivants
  - la condition globale vaut `false`, tout va bien

- dans l'expression : `(((i/j) != 0) && (j != 0))`
  - `(i/j)` est évalué d'abord : j valant 0, cette expression produit un résultat invalide
  - la condition globale devient incohérente (et l'exécution du programme est généralement stoppée immédiatement)

5.3.3 Priorité des opérateurs

TABLE 17 – Priorité des opérateurs de comparaison C

Priorité	Opérateurs
2	!, *not*
...	...
6	$a < b$, $a <= b$, $a > b$, $a >= b$
7	$a == b$, $a! = b$
...	...
11	&&, *and*
12	||, *or*

Source : http://en.cppreference.com/w/c/language/operator_precedence

Conseil
L'utilisation des parenthèses dans les expressions logiques lève tout risque d'ambiguïté.

Par exemple, l'expression logique ($a > b$ *and* $b < c$ *or* $c > d$) peut être interprétée par l'être humain de plusieurs manières (même si l'ordinateur l'analysera mécaniquement de gauche à droite) :

- (($a > b$ *and* $b < c$) *or* $c > d$)
- ou bien ($a > b$ *and* ($b < c$ *or* $c > d$))

La comparaison va être ici exploitée de gauche à droite, mais en profondeur d'abord : les parenthèses permettent ainsi de lever toute ambiguïté et forceront une évaluation sûre de l'expression.

5.3.4 Simplification des expressions logiques

Loi de De Morgan :

- NON (p ET q) est identique à : NON p OU NON q
- NON (p OU q) est identique à : NON p ET NON q

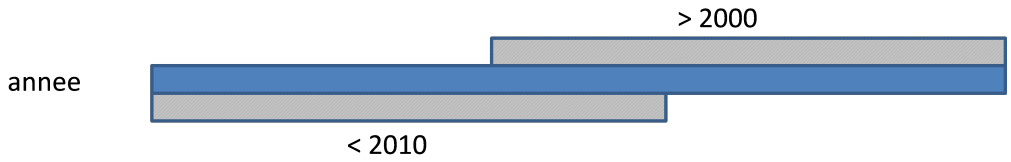
Cf. votre cours de logique.

5.3.5 Expressions logiques mal formées

Certaines expressions logiques peuvent être mal formées :

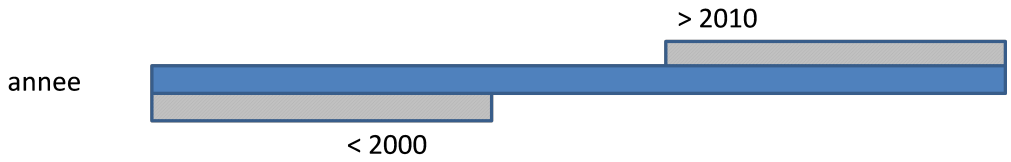
- les *tautologies* (voir Figure 16 en page 61) sont des expressions logiques qui sont toujours vraies, comme par exemple : $((annee > 2000)||(annee < 2010))$

FIGURE 16 – Tautologie



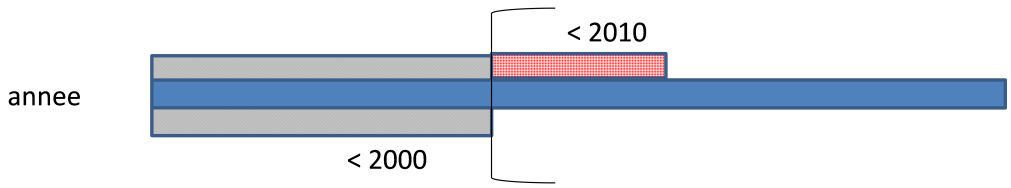
- les *contradictions* (voir Figure 17 en page 61) sont des expressions logiques qui sont toujours fausses, comme par exemple : $((annee < 2000)\&\&(annee > 2010))$; la valeur de *annee* ne peut être à la fois (and) inférieure à 2000 ET supérieure à 2010 ;

FIGURE 17 – Contradiction



- les *redondances* (voir Figure 18 en page 61) sont des expressions logiques dans lesquelles certains éléments sont inutiles, par exemple : $((annee < 2010)\&\&(annee < 2000))$; ici, seule la deuxième partie de l'expression est utile

FIGURE 18 – Redondance



5.3.6 Exercices

1. les variables x, y et z étant des nombres réels, les autres variables étant des nombres entiers, déterminer les types des expressions suivantes :

true	1515	(a == b)	(a - b)
(2+3.14*z)	((i+1)<150)	((a %2) == 0)	false
'a'	((i>10)||((j<12))	"3x2+5x-5/2"	!a
"false"	"hello"	3.1415927	0
1e-15	((100+x)-y)	"12530"	"2.5"
"3.14"	x/2	a/2	2%3

2. L'expression logique équivalente à la proposition suivante "*la valeur a est comprise entre 0 et 20, tous 2 exclus*" est : $(a > 0)$ `and` $(a < 20)$. Écrire les expressions logiques équivalentes aux propositions suivantes (x, y et z étant des nombres entiers) :

   (a) les valeurs de x et y sont supérieures à 5

   (b) les valeurs de x, y et z sont égales

   (c) les valeurs de x et y sont identiques mais différentes de z

   (d) la valeur de x est comprise entre y et z (y<z), mais différente de y et de z

   (e) parmi les valeurs de x, y et z, 2 valeurs sont identiques

   (f) parmi les valeurs de x, y et z, 2 valeurs au moins sont identiques

   (g) x est pair

6 Interactions avec l'utilisateur

6.1 Quelques repères...

- *ecriture* ou *affichage* : opération d'envoi d'un message vers l'environnement pour donner une consigne ou une information
- *lecture* ou *saisie* : opération de récupération d'une information en provenance de l'environnement afin d'influencer le déroulement
- *périphérique d'écriture standard* : périphérique ciblé par l'écriture, l'écran
- *périphérique de lecture standard* : périphérique ciblé par la lecture, le clavier
- *flux* : représentent les canaux de communication avec l'environnement d'un programme :
  - `stdin` : entrée standard du système, le clavier
  - `stdout` : sortie standard du système, l'écran

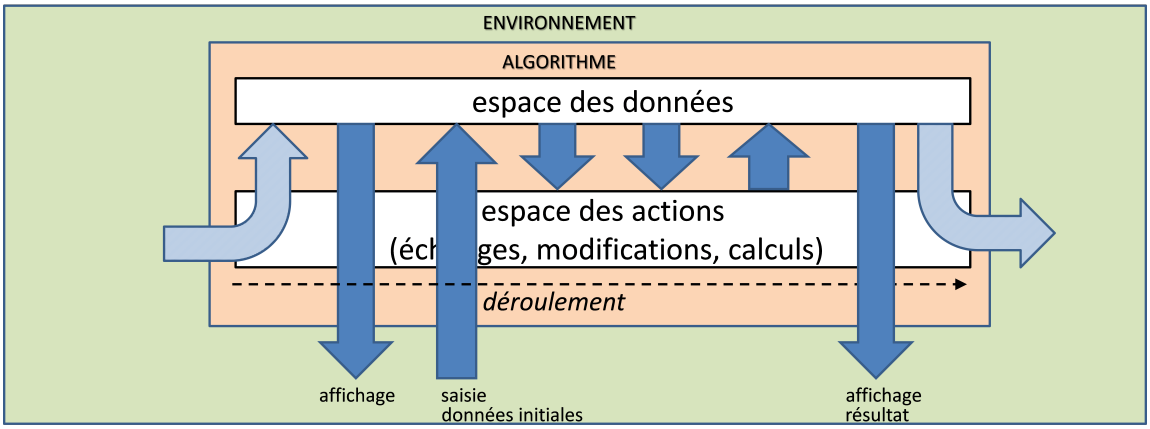
6.2 Lectures / écritures

Les instructions de lecture/écriture permettent les échanges d'un programme avec son environnement, et, plus précisément ici, la description d'un dialogue avec l'utilisateur du programme :

- récupérer les valeurs saisies au clavier
- afficher des valeurs à l'écran (consignes, résultats, etc.)

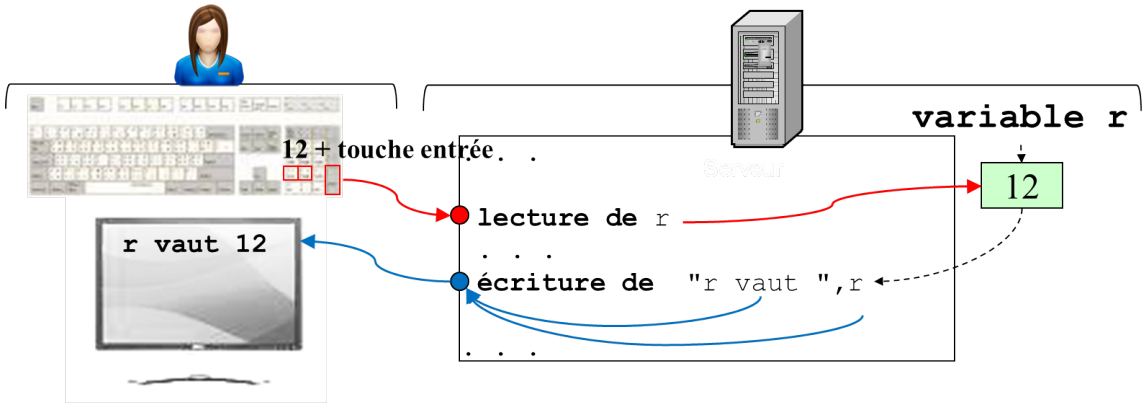
(voir Figure 19 en page 63)

FIGURE 19 – Échange de données avec l'environnement immédiat du programme : affichage et saisie



Ces échanges permettent la construction de programmes répondant à une grande variété de questions, sans à avoir à modifier le programme ou concevoir un nouvel algorithme.

FIGURE 20 – Lecture du clavier et écriture à l'écran : dialogue utilisateur



Algorithme 18 Dialogue somme de 2 nombres

```
1: Variables
2: a : entier ← 0
3: b : entier ← 0
4: r : entier ← 0                          ▷ pour mémoriser la somme de a et b
5: Debut
6: entier r ← 0
7: Ecrire "entrer 2 nombres entiers"       ▷ consigne de saisie
8: Lire a, b                               ▷ récupérer la saisie dans a et b
9: r ← (a + b)                             ▷ calculer r
10: Ecrire "le résultat est ",r            ▷ afficher le résultat, la valeur de r
11: Fin
```

Attention

Les données lues devront toujours être vérifiées avant d'être utilisées dans les instruction suivantes d'un programme

A titre d'illustration, la vérification d'une valeur saisie n, nombre entier, pourrait être réalisée ainsi :

Algorithme 19 vérifier les données lues

```
1: Ecrire "entrer un entier :"
2: Lire n
3: tant que n est incorrect faire     ▷ remplacer la condition selon la vérification souhaitée
4:     Ecrire "Erreur, recommencer !"
5:     Ecrire "entrer un entier :"
6:     Lire n
7: fin tant que
8: ...suite des instructions...       ▷ la valeur de n est correcte
```

Voir section 9 page 82.

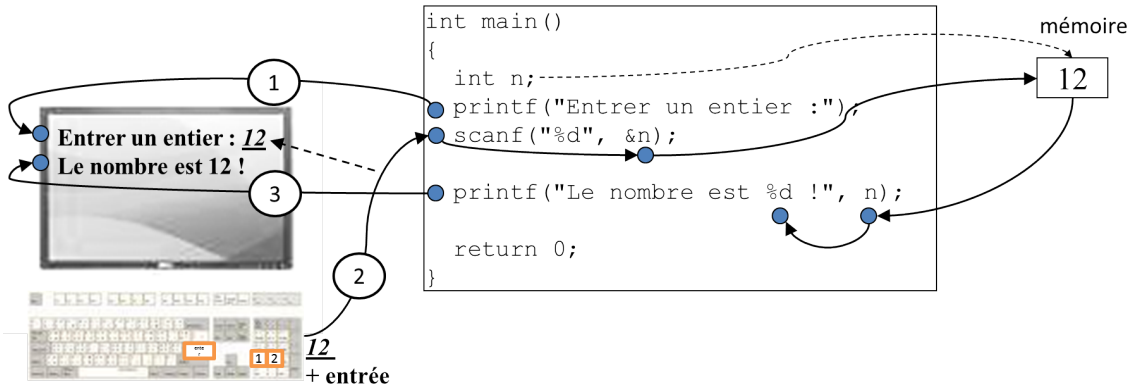
6.3 Lectures / écritures en C

Les instructions de lecture/écriture ne font pas partie du langage C de base mais sont incluses dans une bibliothèque standard.

Il est donc nécessaire d'indiquer au compilateur son utilisation en utilisant la directive `#include <stdio.h>` du préprocesseur.

FIGURE 22 – Demander la saisie d'un nombre entier et afficher sa valeur

FIGURE 21 – Lecture du clavier et écriture à l'écran : dialogue utilisateur en C



Code source 44 – Demander la saisie d'un nombre entier en C

```c
#include <stdio.h>
int main()
{
  int n = 0;
  printf("entrer un entier :");
  scanf("%d", &n);
  printf("le nombre est %d !", n);

  return 0;
}
```

Code source 45 – Demander la saisie d'un nombre entier en C++

```cpp
#include <iostream>
int main()
{
  int n = 0;
  std::cout << "entrer un entier :";
  std::cin >> n;
  std::cout << "le nombre est "
       << n << " !";
  return 0;
}
```

6.3.1 Écriture en C

Plusieurs fonctions permettent l'affichage de valeurs à l'écran :

- `putchar` : affiche un seul caractère, plus précisément le symbole de la table ASCII correspondant à une valeur numérique ;
- `puts` : affiche une simple chaine de caractères puis effectue un saut à la ligne suivante ;
- `printf` : affiche une chaine de caractères qui contient un format avec lequel des valeurs seront fusionnées

Code source 46 – Prototypes des fonctions d'affichage en C

```c
int putchar ( int character );
int puts ( const char * str );
int printf ( const char * format, ... );
```

où

- `puts`, `printf` et `putchar` sont les noms des fonctions,
- *character* : le caractère à afficher

- *str* : la chaine de caractères à afficher
- *format* : la chaine de format utilisée pour l'affichage
- ... : contient les valeurs à insérer dans la chaine de format

Code source 47 – affichage avec la fonction putchar

```
1  char a = 'b';
2
3  putchar('a'); /* affiche le caractère 'a' */
4  putchar(a);  /* affiche la valeur de la variable a */
5  putchar('\n'); /* effectue un saut de ligne : '\n' */
6  putchar(a+1); /* affiche la valeur de l'expression (a+1) comme caractère */
7  putchar(32); /* affiche la valeur 32 (un espace)*/
8  putchar(a+2); /* affiche la valeur de l'expression (a+2) comme caractère */
```

Le résultat est :

```
ab
c d
```

Code source 48 – affichage avec la fonction puts

```
1  char * ch = "Hello\n\tworld !";
2
3  puts("Hello world !");
4  puts(ch);
```

Le résultat est :

```
Hello
world !
Hello world !
```

Code source 49 – Exemple d'affichage en C

```
1 int r = 2;
2 float pi = 3.1415927;
3 char c = 'O';
4 char * d = "disque";
5 printf("le %s de rayon %d et de centre %c\n"
6     " a une aire de %f",
7     d, r, c, pi*r*r);
```

L'affichage obtenu est le suivant :

```
le disque de rayon 2 et de centre O
 a une aire de 12.566371
```

On peut remarquer dans la chaine de format :

- à chaque symbole % correspond l'insertion de la valeur d'une valeur provenant d'une liste de valeurs passées à la fonction `printf`
- après chaque symbole % une lettre indique la nature de la valeur qui sera insérée à cet endroit :
  - `%d` : une valeur numérique entière
  - `%f` : une valeur réelle (`float` ou `double`)
  - `%c` : un caractère
  - `%s` : une chaine de caractères

De plus, on peut remarquer des caractères spéciaux introduits par \ qui participent à la mise en forme de la chaine de sortie :

- `\n` : passage à la ligne suivante
- `\t` : avance d'une tabulation (généralement 4 espaces)

Le format peut inclure d'autres éléments, comme par exemple la taille de la sortie :

```
1 printf("'%4d"  '%8.2f'  '%20s'",12,␣2.5,␣"hello");
```

affiche :

```
'  12' '    2.50' '               hello'
 ....   ..... ..   ....................
```

```
1 printf("'%4d"  '%8.2f'  '%-20s'",-12,␣-2.5,␣"hello");
```

affiche :

```
' -12' '   -2.50' 'hello               '
 ....   ..... ..   ....................
```

De nombreuses autres possibilités sont définies.

6.3.2 Lectures en C

La lecture des données consiste en l'extraction des valeurs saisies au clavier et leur affectation à une variable.

Attention

La lecture des données devra toujours être contrôlée :

- vérifier la valeur retournée par la fonction utilisée
- vérifier la donnée saisie par rapport aux valeurs attendues

(cf section 9.4 page 88)

Trois fonctions principales sont disponibles :

- `getchar` : lecture du prochain caractère de l'entrée standard
- `scanf` : lecture du prochain mot de l'entrée standard (un mot s'arrête au premier caractère 'espace blanc' (en anglais : *whitespace*)) et conversion de la valeur selon un code de format
- `fgets` : lecture d'un certain nombre de caractères d'une ligne (la lecture s'arrête lorsque le caractère de fin de ligne (touche entrée) est rencontrée ou lorsque la variable réceptrice est pleine) ; la fonction `sscanf` y est associée pour analyser la chaine et en extraire les données elle réalise le même traitement que `scanf` mais à partir d'une chaine

Code source 50 – Syntaxe de la saisie en C

```
1 int getchar ( void );
2 int scanf ( const char * format, ... );
3 char * fgets ( char * str, int num, FILE * stream );
4 int sscanf ( const char * s, const char * format, ...);
```

où

- `scanf`, `getchar` et `fgets` : les nom des fonctions de saisie
- *format* : contient le code de conversion de la saisie vers un type de donnée défini
- ... : contient les adresses des variables cibles
- `&` est l'opérateur d'adressage d'une variable (obtient son adresse pour stocker la valeur saisie)
- *str* : le nom de la variable réceptrice, une chaine de caractères
- *num* : le nombre de caractères à lire du flux d'entrée (= taille maximale de *str*)
- *stream* : le flux d'entrée, ici `stdin`
- *s* : la chaine à analyser (elle a été lue par `fgets`)

Code source 51 – Exemple de dialogue avec getchar

```
1 char reponse = ' ';
2 puts("Voulez-vous continuer ? (o/n) :");
3 reponse = getchar();
4 printf("vous avez repondu '%c' !", reponse);
```

Si la saisie effectuée est 5, l'affichage obtenu sera le suivant :

```
Voulez-vous continuer ? (o/n) :
y
vous avez repondu 'y' !
```

Code source 52 – Exemple de dialogue avec scanf

```
1 int a = 0;
2 puts("Entrer un nombre entier :");
3 scanf("%d",&a); // la valeur entiere saisie est stockée à l'adresse de a
4 printf("a vaut %d\n et le double de a vaut %d\n",
5     a,(a*2));
```

Si la saisie effectuée est 5, l'affichage obtenu sera le suivant :

```
Entrer un nombre entier :
5
a vaut 5
 et le double de a vaut 10
```

Code source 53 – Exemple de dialogue avec fgets

```
1 char phrase[40];
2 puts("Entrer une phrase :");
3 fgets(phrase, 39, stdin);
4 printf("la phrase est '%s'",phrase);
```

Si la saisie effectuée est "il faut beau", l'affichage obtenu sera le suivant :

```
Entrer une phrase :
il faut beau
la phrase est 'il faut beau
'
```

Remarque : l'apostrophe est passée à la ligne suivante à cause de la touche *Entrée* (\n) récupérée lors de la saisie.

Code source 54 – Exemple de dialogue avec fgets et sscanf

```
1 char texte[256];
2 int a = 0;
3 puts("Entrer un nombre entier :");
4 fgets(texte, 255,stdin );
5 sscanf(texte, "%d", &a); // la valeur entiere saisie est stockée à l'adresse
6 printf("a vaut %d\n et le double de a vaut %d\n",
7     a,(a*2));
```

Si la saisie effectuée est 5, l'affichage obtenu sera le suivant :

```
Entrer un nombre entier :
5
a vaut 5
 et le double de a vaut 10
```

D'autres possibilités sont disponibles (cf. norme)

6.4 Analyse du buffer d'entrée avec scanf

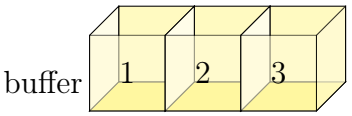
Le *buffer* est une zone de mémoire intermédiaire. Un buffer d'entrée est une mémoire intermédiaire utilisée lors de la saisie d'informations au clavier.

Au fur et à mesure de la saisie des caractères, ces derniers s'accumulent dans le buffer et ne sont pris en compte par les instructions de saisie qu'une fois la touche 'Entrée' frappée.

Par exemple, soit le programme suivant qui demande la saisie d'un nombre puis d'un caractères :

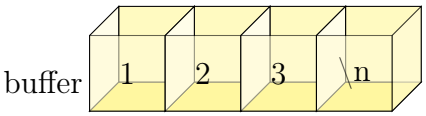
```
1  #include <stdio.h>
2  int main()
3  {
4          int i;
5          char ch[80];
6          printf("entrer un entier : ");
7          scanf("%d", &i);
8          printf("entrer une chaine : ");
9          fgets(ch, sizeof ch, stdin);
10         printf("Vous avez saisi %d et \"%s\"\n", i, ch);
11
12         return 0;
13 }
```

Par exemple, la saisie des chiffres 1, 2 puis 3 s'accumulent dans le buffer :



L'instruction de saisie est toujours en attente.

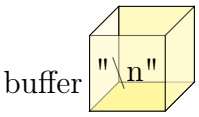
L'utilisateur frappe ensuite la touche entrée : le caractère '\n' est ajouté au buffer :



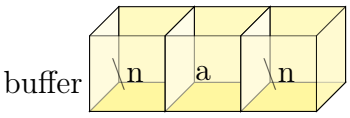
puis déclenche l'analyse du buffer par la fonction `scanf` :

- le format '%d' indique qu'un entier doit être extrait
- le caractère 1 est un entier ; 2 et 3 le sont aussi
- le caractère suivant '\n'est pas un chiffre : la fonction `scanf` a localisé le nombre 123 qu'elle mémorise à l'adresse de la variable spécifiée, supprime les 3 chiffres du buffer, puis l'exécution reprend à l'instruction qui suit jusqu'à la prochaine saisie

Après la saisie, le buffer contient :



La saisie d'un caractère ajoute une lettre au buffer et la frappe de la touche entrée ajoute ce caractère également :



6.4.1 Gestion des espaces blancs

Les espaces blancs (en anglais : *whitespace*) (espace, tabulation, entrée) sont des caractères ignorés lors de l'analyse du buffer d'entrée par la fonction `scanf` :

- les formats '%d', '%f', '%s' ignorent les caractères blancs qui précédent le 1er caractère non blanc ; l'analyse du buffer est poursuivie jusqu'au 1er caractère non blanc, ce dernier étant laissé dans le buffer
- si un caractère espace est inclus dans le format, il représente un certain nombre d'espaces blancs à ignorer avant l'analyse du buffer

6.4.2 Saisie contrôlée avec `sscanf`

La fonction `sscanf` est proche de la fonction `scanf` ; elle permet d'analyser une chaine de caractère et retourne le nombre de valeurs extraites.

Cela permet de dissocier l'opération de lecture de celle de l'extraction des valeurs.

```
1 char ligne[4096];
2 if (fgets(line, sizeof(ligne), stdin) != 0)
3 {
4   ...décoder la ligne avec sscanf ...
5 }
```

6.5 Exercices

1. Proposer l'algorithme d'un dialogue permettant le calcul du volume d'un pavé creux à partir de sa longueur, sa largeur, sa hauteur et son épaisseur.

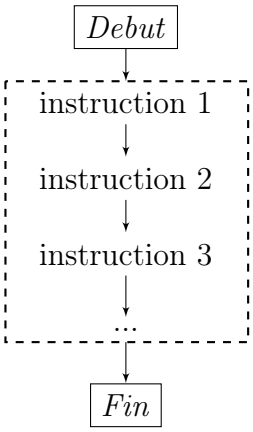
Quatrième partie

Instructions de contrôle

7 Séquence et bloc d'instructions

La séquence des instructions d'un algorithme se déroule linéairement. Chaque opération n'est exécutée que lorsque la précédente est terminée. A la fin de l'exécution, toutes les instructions auront été exécutées. Elles forment un bloc.

FIGURE 23 – Séquence d'actions ou bloc



7.1 Bloc d'instructions en C

En langage C, un bloc d'instructions est encadré par un jeu d'accolades. Il peut comporter des déclarations qui resteront locales au bloc.

7.2 Ruptures de séquence

Il arrive cependant que certaines instructions soient à exécuter seulement dans certains cas, ou que, pour d'autres, on doive en répéter l'exécution un certain nombre de fois.

Les *instructions de contrôle* (ou *structures de contrôle*) vont permettent la création d'une rupture dans l'exécution en séquence des actions d'un algorithme et prendre en compte ces cas et en contrôler la bonne exécution.

8 Exécution conditionnelle, décision : "Si alors sinon", "Selon"

Exemple en mathématique :

1. Déterminer si un nombre `a` est multiple d'un autre nombre `m`
   - $a \in \mathbb{N}^*$
   - $m \in \mathbb{N}^*$

- $f(a, m) \in \{Vrai, Faux\}$
- $f(a, m) = \begin{cases} Vrai & \text{si le reste de } a/m = 0 \\ Faux & \text{sinon} \end{cases}$

En algorithmique : il y a exécution conditionnelle lorsque, dans une séquence, un bloc d'actions peut ne pas être pas exécuté.

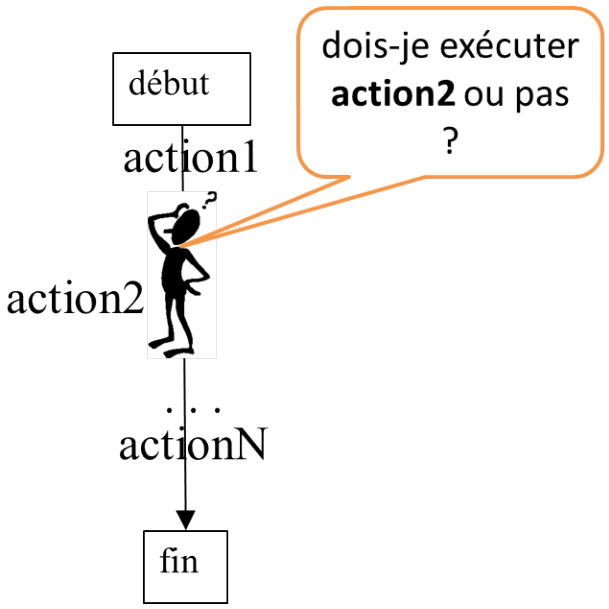
Algorithme 20 Déterminer si un nombre `a` est multiple d'un autre nombre `m` après-midi

```
1: Fonction F(a : entier , m : entier) : booléen          ▷ a est-il multiple de m ?
Pre-condition: ((m <> 0))
2:     si ((a mod m = 0)) alors
3:         return Vrai
4:     sinon
5:         return Faux
6:     fin si
7: fin Fonction
```

8.1 Sans alternative : si ... alors ... finSi, `if ()`

La structure de contrôle conditionnelle SI permet de choisir d'exécuter ou pas une séquence d'actions selon qu'une condition est vraie ou pas. Si la condition est vraie, le bloc d'actions est exécuté, sinon aucune action n'est exécutée (voir Figure 25 en page 74) et on passe à l'instruction qui suit la structure conditionnelle.

FIGURE 24 – Exécution conditionnelle



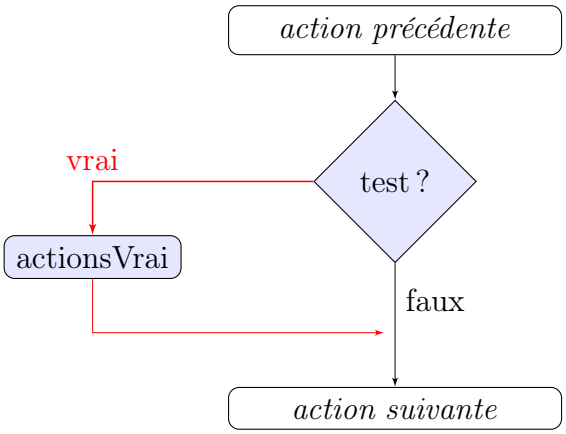
Algorithme 21 Structure conditionnelle

```
1: ...actions précédentes...
2: si exprLog alors                          ▷ test est une condition
3:     actionsVrai                           ▷ actionsVrai réalisé seulement si test vaut vrai
4: fin si
5: ...actions suivantes...
```

où ;

- *exprLog* : expression logique qui exprime la condition de l'exécution
- *actionsVrai* : bloc d'actions réalisé seulement si *exprLog* vaut vrai

FIGURE 25 – Le bloc actionsVrai est réalisé si test vaut vrai



Syntaxe C 8.12 Structure de contrôle conditionnelle simple

```
if (exprLog) {
... bloc ...
}
```

où :

- `if` : mot-clef introduisant une structure de controle conditionnelle
- *exprLog* : expression logique évaluée pour choisir d'exécuter le bloc d'instruction ou pas
- *bloc* : entre accolades, bloc d'instructions qui sera exécuté si *exprLog* vaut `true`

Code source 55 – Exemle de structure conditionnelle

```
1 const int MAJORITE = 18;
2 int age;
3 scanf("%d", &age);
4 if (age >= MAJORITE) {
5         printf("majeur !");
6 }
```

Remarque : une version plus efficace pourrait être majeur = (age >= majo).

8.2 Avec alternative : si ... alors ... sinon ... finSi, `if () else`

La structure de contrôle conditionnelle *avec alternative* permet d'exécuter un bloc d'instructions si une condition est vraie et un autre bloc (c'est une alternative) dans le cas contraire (voir Figure 26 en page 75)

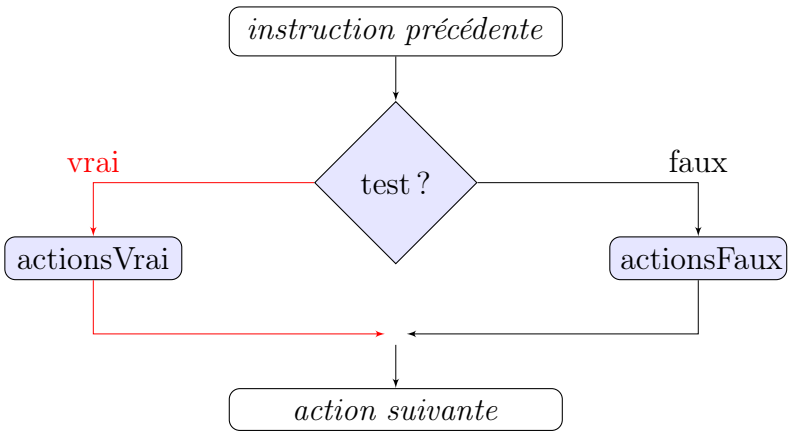
Algorithme 22 Structure conditionnelle avec alternative

```
1: si exprLog alors                          ▷ exprLog est une condition
2:     actionsVrai                           ▷ si test est vrai
3: sinon
4:     actionsFaux                           ▷ si test n'est pas vrai
5: fin si
```

où ;

- *exprLog* : expression logique qui exprime la condition qui va être évaluée
- *actionsVrai* : bloc d'instruction réalisé seulement si *exprLog* vaut vrai
- *actionsFaux* : bloc d'instruction réalisé sinon

FIGURE 26 – L'un des 2 blocs, actionsVrai ou actionsFaux, sera réalisé



Syntaxe C 8.13 Structure de contrôle conditionnelle avec alternative

```
if (exprLog) {
... bloc1 ...
} else {
... bloc2 ...
}
```

où :

- `if` : mot-clef introduisant une structure de controle conditionnelle

- *exprLog* : expression logique évaluée pour choisir d'exécuter le bloc d'instruction 1 ou le bloc d'instructions 2
- *bloc1* : entre accolades, bloc d'instructions qui sera exécuté si *exprLog* vaut `true`
- `else` : mot-clef introduisant l'alternative d'une structure de controle conditionnelle
- *bloc2* : entre accolades, bloc d'instructions qui sera exécuté si *exprLog* vaut `false`

Code source 56 – Exemple de structure conditionnelle avec alternative

```
1 const int MORITE = 18;
2 int age;
3
4 scanf("%d", &age);
5 if (age >= MAJORITE) {
6         printf("majeur !");
7 } else {
8         printf("pas encore majeur !");
9 }
```

8.3 Avec alternative multiples

La structure de contrôle conditionnelle avec *alternatives multiples* permet d'effectuer une suite de tests et de choisir une séquence d'instructions à réaliser.

Algorithme 23 Structure conditionnelle multiples

```
1: si exprLog1 alors
2:     actions1                                   ▷ exécuté si exprLog1 est vrai
3: sinon si exprLog2 alors
4:     actions2                          ▷ exécuté si exprLog2 est vrai (exprLog1 est faux)
5: sinon si exprLog3 alors
6:     actions3              ▷ exécuté si exprLog3 est vrai (exprLog1 et exprLog2 sont faux)
7: sinon si etc. alors
8:     etc.
9: sinon                                  ▷ si aucun des tests précédents n'était vrai
10:    actions                                          ▷ tous les tests sont faux
11: fin si
```

Syntaxe C 8.14 Structure de contrôle conditionnelle avec alternatives multiples

```
if (exprLog1) {
... bloc1 ...
} else if (exprLog2) {
... bloc2 ...
} else if (exprLog3) {
```

```
... bloc3 ...
} else {
... bloc4 ...
}
```

où :

- `if` : mot-clef introduisant une structure de controle conditionnelle
- *exprLog1*, *exprLog2*, *exprLog2* : expressions logiques évaluées qui conditionnent l'exécution de l'un des bloc correspondants, *bloc1* ou *bloc2* ou *bloc3*

Code source 57 – Exemple de structures alternatives multiples en C

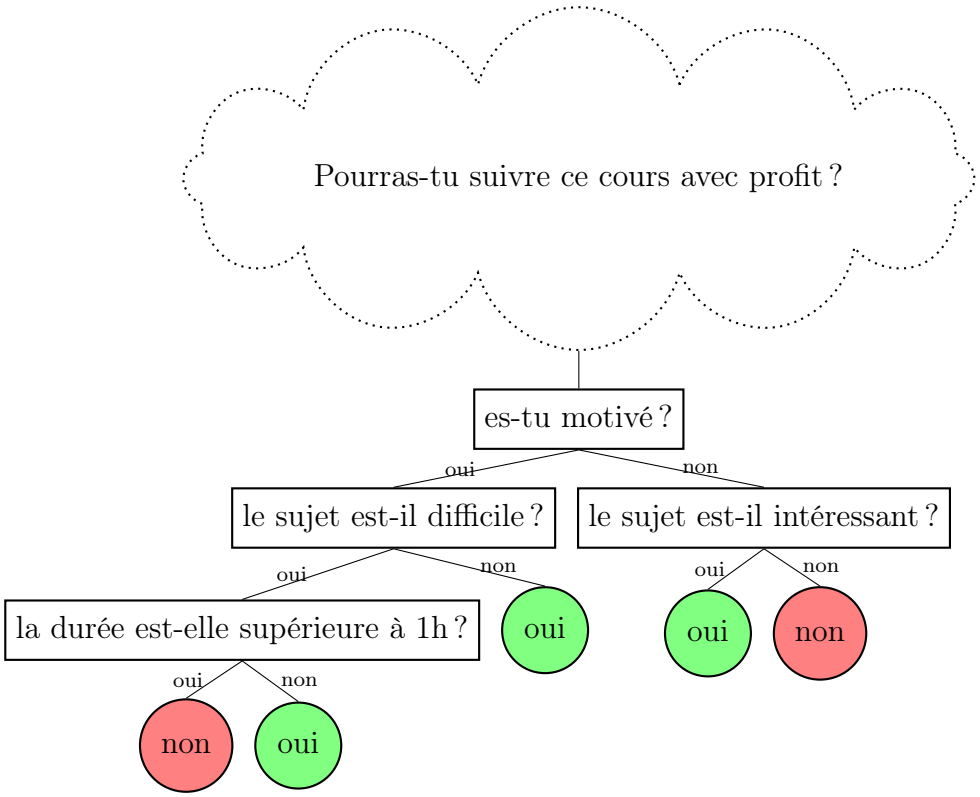
```
1  int getTrimestre(int mois)
2  {
3          assert((mois >= 1) && (mois <= 12))
4          int numTrim = 0;
5          if ((mois == 9) or (mois == 10 ) or (mois == 11)) {
6           numTrim = 1;
7          }
8          else if ((mois == 12) or (mois == 1 ) or (mois == 2)) {
9           numTrim = 2;
10         }
11         else if ((mois == 3) or (mois == 4 ) or (mois == 5)) {
12          numTrim = 3;
13         }
14         else {
15          numTrim = 4;
16         }
17         return numTrim;
18 }
```

8.4 Structures conditionnelles imbriquées

8.4.1 Exemple : arbre de décision

Dans le cas de traitement de décisions plus complexes, des structures conditionnelles peuvent être imbriquées.

Par exemple : comment prévoir si oui ou non tu pourras suivre ce cours (où n'importe quel autre) avec profit ?



Dans les blocs d'une structure conditionnelle, on peut à nouveau introduire une nouvelle structure conditionnelle (et ainsi de suite) : les structures sont dites *imbriquées*.

Algorithme 24 Structures conditionnelles imbriquées

```
1:  si exprLog1 alors
2:      si exprLog2 alors
3:          actions                          ▷ exprLog1 est vrai et exprLog2 est vrai
4:      sinon
5:          actions                          ▷ exprLog1 est vrai et exprLog2 est faux
6:      fin si
7:  sinon                                    ▷ exprLog1 n'est pas vrai
8:      si exprLog3 alors
9:          actions                          ▷ exprLog1 est faux et exprLog3 est vrai
10:     sinon
11:         actions                          ▷ exprLog1 est vrai et exprLog3 est faux
12:     fin si
13:     ... etc. ....
14: fin si
```

Code source 58 – Structures conditionneles imbriquées alternative en C/C++

```
1  if (exprLog1)
2  { // bloc d'instructions à exécuter
3   // si exprLog1 vaut true
4           if (exprLog2)
5           { // bloc d'instructions à exécuter
6            // si exprLog2 vaut true (exprLog1 vaut true également)
7                   ...
```

```
8           }
9           else
10          { // bloc d'instructions à exécuter
11           // sinon (exprLog2 vaut false, exprLog1 vaut true)
12                  ...
13          }
14  }
15  else
16  { // bloc d'instructions à exécuter
17   // sinon
18          if (exprLog3)
19          { // bloc d'instructions à exécuter
20           // si exprLog3 vaut true (exprLog1 vaut false)
21                  ...
22          }
23          else
24          { // bloc d'instructions à exécuter
25           // sinon (test3 vaut false, exprLog1 également)
26                  ...
27          }
28  }
```

8.5 Structures de choix multiple

Il arrive qu'un traitement conditionnel s'appuie sur le test successif de plusieurs valeurs d'une même variable.

Algorithme 25 Structures conditionnelles successives

```
1: si idVar = valeur1 alors
2:     actions1
3: sinon si idVar = valeur2 ou idVar = valeur3 alors
4:     actions2
5: sinon
6:     actionsParDefaut
7: fin si
```

Code source 59 – Structures conditionnelles imbriquées en C

```
1  if (idVar == valeur1) {
2    // action1
3  } else if ((idVar == valeur2)||(idVar == valeur3)) {
4    // action2
5  } else {
6    // actionsParDefaut
7  }
```

La structure de contrôle *selon* permet de simplifier l'écriture.

8.5.1 Structures de choix multiple avec "Selon", `switch`

La structure de choix multiple est une écriture plus compacte des choix multiples d'une même variable.

Selon les différents cas des valeurs d'une expression, orienter vers l'un ou l'autre traitement.

Algorithme 26 Structures conditionnelles successives

```
1: selon expr faire
2:     cas valeur1
3:         actions1
4:     cas valeur2, valeur3
5:         actions2
6:     par defaut
7:         actionsParDefaut
8: fin selon
```

La structure de contrôle C `switch` permet le test la valeur d'une variable et oriente le traitement selon cette valeur : plusieurs cas sont donc définis.

Syntaxe C 8.15 Structure de contrôle conditionnelle choix multiple

```
switch (expr) {
    case const1 :
        bloc1 ;
        break ;
    case const2 :
    case const3 :
        bloc2 ;
        break ;
    default :
        blocParDefaut ;
}
```

où :

- `switch` : mot-clef introduisant une structure de choix multiple
- *expr* : est une expression numérique entière
- `case` : instruction qui introduit un cas de test en précisant la valeur de l'expression à tester pour ce cas
- *const1*, *const2*, etc. : les valeurs constantes de cette expression testées pour chacun des cas

- *bloc1*, *bloc2*, etc. sont les traitements effectués selon les cas
- `break` : quitte le traitement d'un cas et reprend l'exécution à l'instruction qui suit la structure de contrôle
- `default` : instruction qui introduit les instructions(*blocParDefaut*) à réaliser si aucun cas n'a été sélectionné

Ainsi la structure imbriquée proposée dans la partie I, s'écrit plus simplement :

Code source 60 – Exemple de structures alternatives multiples en C

```
1  int getTrimestre(int mois)
2  {
3          assert((mois >= 1) && (mois <= 12))
4          int numTrim = 0;
5          switch (mois) {
6                  case 9:
7                  case 10:
8                  case 11:
9                          numTrim = 1;
10                         break;
11                 case 12:
12                 case 1 :
13                 case 2 :
14                         numTrim = 2;
15                         break;
16                 case 3 :
17                 case 4 :
18                 case 5 :
19                  numTrim = 3;
20                  break;
21                 default :
22                  numTrim = 4;
23         }
24         return numTrim;
25 }
```

8.6 Quand utiliser une structure conditionnelle ?

On utilise une structure conditionnelle quand une instruction, ou un bloc d'instructions, n'est pas toujours exécuté, et quand il est exécuté, il ne l'est qu'une seule fois ; son exécution dépend d'une condition :

- dans le cas d'une structure conditionnelle simple (*si ... alors ...actions... finSi*, le bloc d'instructions sera exécuté une seule fois ou ne sera pas du tout exécuté.
- dans le cas d'une structure conditionnelle avec alternative (*si ... alors ...actions1 sinon ...actions2... finSi*, un seul des 2 blocs sera exécuté et il le sera une seule fois.

Conseil

Dans les structures conditionnelles, un seul bloc d'instruction sera exécuté[a].

[a] une exception peut être faire pour la structure *selon* en C, `switch`, pour laquelle, dès qu'on entre dans un cas, les blocs sont exécutés jusqu'à ce qu'on rencontre l'instruction `break`. En tout cas, les blocs ne seront exécutés qu'une seule fois.

Attention

La condition de contrôle d'une structure conditionnelle doit être évaluable, c'est-à-dire que les éléments qui la composent devront avoir été initialisés auparavant.

8.7 Exercices

1. Proposer les algorithmes de fonctions répondant aux questions suivantes :
   - Déterminer la plus petite valeur de 2 entiers
   - Déterminer le nombre de racines d'un polynôme du 2nd degré
   - Déterminer le texte correspondant à l'état de l'eau : "glace" si la température est inférieure à 0°C, "eau" si la température est supérieure à 0°C et inférieure à 100°C, "vapeur" si la température est supérieure ou égal à 100°C
   - Déterminer si une date est valide : une date est représentée par 3 entiers, jour, mois et année
2. Proposer le dialogue mettant en oeuvre l'arbre de décision précédant (voir Figure 8.4.1 en page 77)

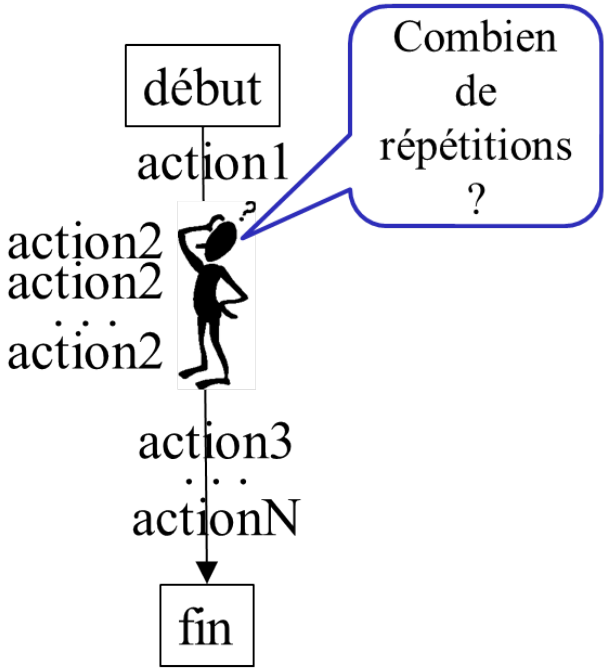
9 Exécution répétée : "Tant que", "Pour"

Les *structures itératives* ou *répétitives* (ou *boucles*, (en anglais : *loop*)) permettent l'exécution répétée d'un bloc d'instructions. Il sera important de déterminer correctement le nombre de répétition.

Itérations en mathématique

1. Une version de l'algorithme d'Euclide (calcul du Plus Grand Commun Diviseur), consiste à effectuer une suite de divisions euclidiennes. Par exemple, pour le calcul du PGCD de a et b (a étant >= b) :
   - on effectue la division euclidienne de a par b et on note r comme étant le reste
   - ensuite, a prend la valeur de b, b prend la valeur de r,
   - on *recommence* le calcul tant que le reste est différent de 0
   - le PGCD est le dernier reste non nul

FIGURE 27 – Répétition d'un bloc d'action



2. Pour réaliser la somme des nombres entiers de 1 à 5, on doit faire intervenir une succession de sommes :

   Description formelle de la résolution :

   - $i \in \mathbb{N}^*$, $S \in \mathbb{N}^*$
   - $S = \sum_{i=1}^{5} i$

3. Calculer la somme des nombres d'une suite croissante de nombres entiers entre a et b (dans l'intervalle $[a, b]$) multiples d'un nombre m. de manière plus formelle :

   - $a \in \mathbb{N}^*$, $b \in \mathbb{Z}^*$, $a <= b$
   - $m \in \mathbb{N}^*$
   - $S(a, b, m) \in \mathbb{N}$
   - $S = \sum_{i=a}^{b} f(i)$
   - $avec : f(i) = \begin{cases} i & \text{si le reste de } i/m = 0 \\ 0 & \text{sinon} \end{cases}$

4. Exemple de la vie "courante" : remplir une seau avec un verre (remarque : le seau risque de déborder...)

Algorithme 27 Exemple de structure itérative : remplir une seau avec un verre

```
1: le seau est vide
2: tant que le seau n'est pas plein faire
3:     verser un verre d'eau dans le seau
4: fin tant que
```

9.1 Nombre de répétitions indéterminé : "Tant que", `while`

La boucle *tant que* permet l'exécution d'un bloc d'instructions tant qu'une condition est vraie. Le corps de la boucle sera exécuté 0 à N fois.

Corps de la boucle

Le *corps de la boucle* correspond au bloc d'instructions à répéter.

Algorithme 28 Structure itérative : tant que

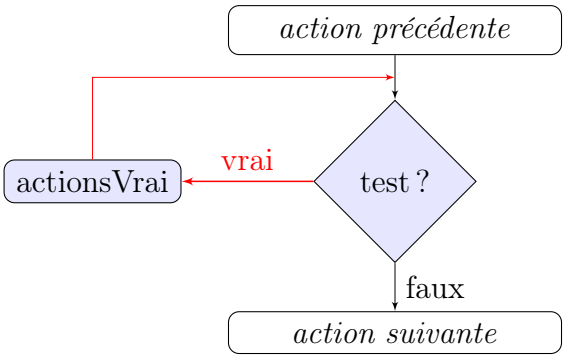
```
1: tant que exprLog faire
2:     actionsVrai          ▷ bloc d'instructions à répéter tant que exprLog vaut vrai
3: fin tant que
```

ou :

- `exprLog` : expression logique qui exprime la condition de répétition des instructions (si elle est vraie et tant qu'elle reste vraie)
- `actionsVrai` : bloc d'instructions à répéter

soit : "Tant que *exprLog* vaut vrai, répéter l'exécution du bloc d'instructions *actionsVrai*". (voir Figure 28 en page 84).

FIGURE 28 – Tant que test vaut vrai, le bloc actionsVrai sera réalisé



Syntaxe C 9.16 Structure de contrôle itérative : `while`

```
while (exprLog) {
... bloc ...
```

FIGURE 29 – Structure de contrôle : répétitive ou itérative

Code source 61 – code C

```
7  #define CAPAMAX 1.5
8  #define CAPAVER 0.25
9  double rb = 0; /* quantite déjà mise en bouteille */
10 /* tant que le remplissage est inférieur
11    à la capacité maximale de la bouteille,
12    ajouter un verre à la bouteille */
13 while (rb < CAPAMAX) {
14   rb = rb + CAPAVER;
15 }
16 /* la bouteille est remplie
17    ... un peu trop... */
```

```
}
```

où :

- `while` : instruction introduisant une structure itérative *tantque*
- *exprLog* : expression logique qui détermine l'exécution du bloc
- *bloc* : bloc d'instruction exécuté si *exprLog* vaut `true` et tant qu'elle vaut `true`

9.1.1 Exemple

Il s'agit de remplir une bouteille de 1.5 litres avec des verres d'une contenance de 0.25 litres

Attention

La condition (s) va être évaluée avant chaque itération : les éléments qui constituent cette condition doivent avoir été initialisés auparavant ET devront être à nouveau modifiées dans le corps de la boucle (sinon on obtient une boucle infinie !)

9.2 Nombre de répétitions déterminé : "Pour", `for`

La boucle *Pour* permet l'exécution d'un bloc d'instructions en faisant varier une variable d'une valeur de début, jusqu'à une valeur de fin, en la faisant varier d'un certain pas (voir figure 30 page 86).

Elle est généralement utilisée lorsque le nombre d'itérations est déterminée : effectuer un traitement pour une suite déterminée de valeurs (parcours systématique des éléments d'un tableau, etc.).

Algorithme 29 Structure itérative : pour

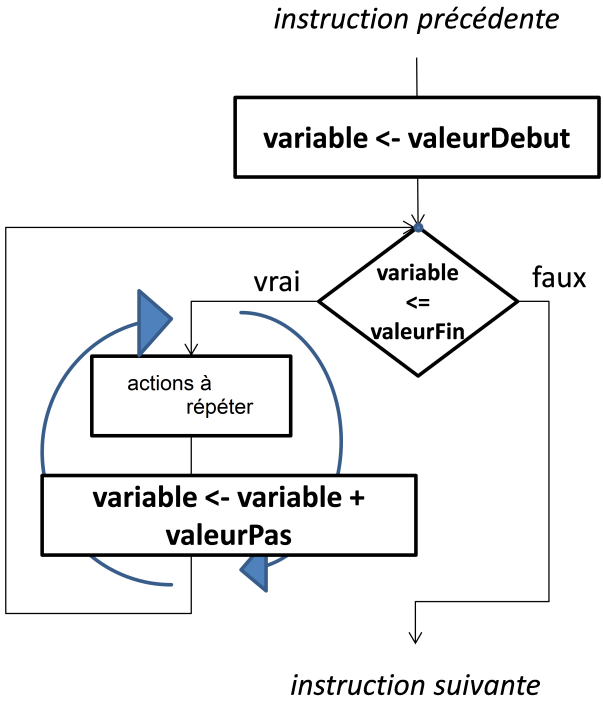
```
1: pour idVar variant de A à B par pas de C faire
2:     actionsARepeter
3: fin pour
```

ou :

- `idVar` : variable qui va permettre le comptage des itérations (c'est la "variable de contrôle de la boucle")
- `A` : valeur initiale affectée à variable en début de boucle
- `B` : valeur finale de la variable : tant que la variable n'a pas atteint cette valeur, on répétera le bloc d'actions
- `C` : incrément de idVar à la fin de chaque itération
- `actionsARepeter` : bloc d'instructions à répéter, dans lequel on utilisera, généralement la valeur de la variable

soit : "Pour la valeur de *idVar* commençant à A, tant qu'elle n'a pas dépassé la valeur B, exécuter le bloc d'instructions *actionsARepeter*, puis ajouter C à *idVar* et comparer à nouveau *idVar* à B pour déterminer si la répétition continue" (C est ici positif).

FIGURE 30 – Exécution répétée : POUR



La structure de contrôle `for` met en oeuvre la structure algorithmique 'pour' :

Syntaxe C 9.17 Structure de contrôle itérative : `for`

for (init ; exprLog ; incr) {
... bloc ...
}

où :

- `for` : instruction introduisant une structure itérative *pour*
- *init* : initialisation ou déclaration+initialisation (**C99**) de la variable de boucle
- *exprLog* : expression logique qui détermine l'exécution du bloc et sa répétition
- *incr* : incrémentation systématique de la variable de boucle à la fin de chaque itération
- *bloc* : entre accolades, bloc d'instruction exécuté si *exprLog* vaut `true` et tant qu'elle vaut `true`

Code source 62 – Exemple de boucle 'for' en C

```
18 #define CAPAMAX 1.5
19 #define CAPAVER 0.25
20 ...
21 float bouteille = 0;
22 int nb = 0; /* nombre de verres */
23 int ct = 0; // compteur de verres
24 /* déterminer le nombre de varres */
25 nb = CAPAMAX / CAPAVER;
26 /* pour le compteur de verres allant de 1 à nb
27    ajouter un verre */
28 for (ct = 1; ct <= nb; ++ct) {
29   bouteille = bouteille + CAPAVER;
30 }
31 /* la bouteille est remplie */
```

La version **C99**autorise la déclaration locale de la variable de boucle ; celle-ci n'est donc pas disponible hors du bloc répété.

Code source 63 – Exemple de boucle 'for' en C (**C99**)

```
32 #define CAPAMAX 1.5
33 #define CAPAVER 0.25
34 ...
35 float bouteille = 0;
36 int nb = CAPAMAX; / CAPAVER /* nbre de verres */
37 /* pour le compteur allant de 1 à nb
38    ajouter un verre */
39 for (int ct = 1; ct <= nb; ++ct) {
40   bouteille = bouteille + CAPAVER;
41 }
42 /* la bouteille est remplie */
```

9.3 Equivalence `while` / `for`

FIGURE 31 – Comparaison des structures de contrôle `while` et `for`

Code source 64 – while

```
1 initialisation
2 while (exprLog) {
3   ...bloc à repeter...
4   increment
5 }
```

Code source 65 – for

```
1
2 for (initialisation; exprLog; increment
3   ...bloc à repeter...
4 }
```

Code source 66 – while

```
1 int i = 0;
2 i = 1;
3 while (i <= 12) {
4   printf("␣%d", i);
5   i = i + 2;
6 }
```

Code source 67 – for

```
1 int i = 0;
2 while (i = 1; i <= 12; i = i + 2) {
3   printf("␣%d", i);
4 }
```

Résultat :

```
1 3 5 7 9 11
```

9.4 Quand utiliser une structure itérative ?

On utilise une structure itérative quand un bloc d'instructions doit être répété un certain nombre de fois : peut-être aucune, 1, 2, ou plus.

9.4.1 Exemple 1

Par exemple, il s'agit de calculer la somme des nombres entiers de 1 à n (de manière triviale...) :

Algorithme 30 Calculer la somme des nombres de 1 à n : comment faire ?

1: s : entier $\leftarrow 0$ ▷ s : accumulateur pour la somme, calculé
2: $s \leftarrow s + 1$
3: $s \leftarrow s + 2$
4: $s \leftarrow s + 3$
5: ...???...comment faire ?

On se rend compte qu'à chaque instruction, on applique le même calcul, mais la valeur accumulée varie selon une suite de raison 1. On peut donc utiliser la structure itérative pour calculer les valeurs successives de cette suite.

Code source 68 – Calculer la somme des nombres de 1 à n en C

```
15         /* DECLARATIONS, INITIALISATIONS */
16         int terme = 0;
17         int somme = 0;
```

```
18          /* TRAITEMENT */
19      /* initialiser la sequence des nombres */
20    terme = 1; /* 1er terme */
21    /* tant qu'on n'a pas dépassé le dernier terme
22       Additionner chaque terme à la somme */
23    while (terme <= n)
24    {
25      /* additionner le terùe à la somme */
26      somme = somme + terme;
27      /* passer au terme suivant de la suite */
28      terme = terme + 1;
29    }
```

9.4.2 Exemple 2

Il s'agit ici, dans un dialogue, de contrôler qu'une valeur saisie est correcte, ici comprise entre 2 bornes : on va demander la saisie d'un nombre, et tant qu'il n'est pas correct (ici compris entre 2 bornes), on va redemander la saisie du nombre :

Algorithme 31 Dialogue de saisie vérifiée d'un nombre , version initiale

```
1: nombre : entier ← 0                          ▷ nombre : valeur à contrôler, saisi
2: Lire nombre
3: si nombre n'est pas correct alors
4:     Lire nombre
5:     si nombre n'est pas correct alors
6:         Lire nombre
7:     fin si
8:     si nombre n'est pas correct alors
9:         Lire nombre
10:    fin si
11:    comment faire ? on ne sait pas combien de fois l'utilisateur va se tromper...
12: fin si
```

On se rend compte de la répétition de saisie/vérification.

Algorithme 32 Dialogue de saisie vérifiée d'un nombre , version initiale

```
1: nombre ! entier ← 0                          ▷ nombre : valeur saisie à contrôler
2: Lire nombre
3: tant que nombre n'est pas correct faire
4:     Ecrire "Incorrect, recommencer !"
5:     Lire nombre
6: fin tant que
```

Code source 69 – Dialogue de saisie vérifiée d'un nombre en C

```
14          /* DECLARATIONS, INITIALISATIONS */
15          const int MINI = 0,
```

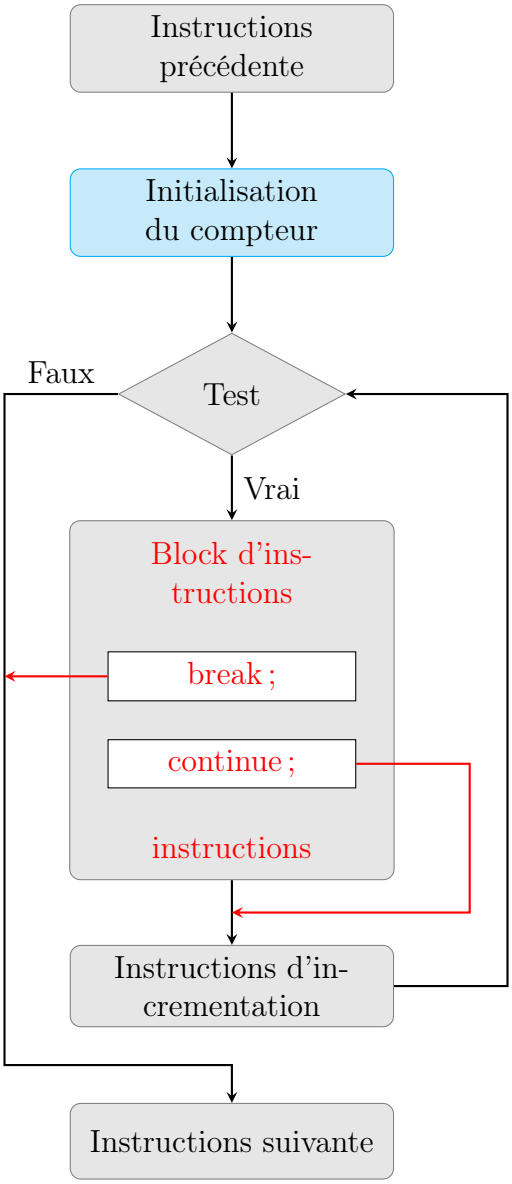
```
16          MAXI = 20;
17            int nombre = 0;
18            /* TRAITEMENT */
19        /* demander la saisie d'un nombre */
20            printf("entrer un nombre compris entre %d et %d :", MINI, MAXI);
21            scanf("%d", &nombre);
22      /* tant que le nombre n'est pas correct,
23         afficher un message et redemander la saisie */
24      while (!((nombre >= MINI) && (nombre <= MAXI))) {
25        printf("ERREUR DE SAISIE, recommencer :");
26        scanf("%d", &nombre);
27      }
28            /* RESULTAT */
29            printf("le nombre %d est correct", nombre );
```

9.5 break et continue

Deux instructions sont associées aux structures de contrôles itératives :

- `break` : quitte la structure de contrôle et poursuit l'exécution à l'instruction suivante
- `continue` : saute à l'itération suivante
  - dans une boucle `for` : l'itération suivante correspond à l'incrémentation de la variable de boucle, avant de tester la nouvelle valeur de celle-ci
  - attention, dans une boucle `while` : l'itération suivante correspond au test de la valeur de la variable de boucle ; si cette dernière n'a pas été incrémentée auparavant, on obtient une boucle infinie !

FIGURE 32 – Instructions `break` et `continue` dans une structure `for`



Source : https://tex.stackexchange.com

9.6 Exercices

1. Utiliser la trace d'exécution pour déterminer les valeurs de a, b et c après exécution de la séquence d'instructions suivante :

```
1  int a = 0,
2          b = 0,
3          c = 0;
4  a = 4;
5  b = a * 2;
6  while (a < b)
7  {
8          c = c + a;
9          a = a + 1;
10 }
```

2. Cet algorithme est-il correct ? Utiliser une trace d'exécution pour le vérifier.

Code source 70 – Exemple de remplissage d'une bouteille avec un verre en C/C++

```
1 const double CAPAMAX = 1.5;
2 const double CAPAVER = 0.33;
3 double rb = 0; /* quantite déjà mise en bouteille */
4 while (rb < CAPAMAX)
5 {
6         rb = rb + CAPAVER;
7 }
```

3. Proposer l'algorithme d'une fonction permettant le calcul du produit de 2 nombres entiers naturels en utilisant l'opérateur d'addition seulement.

4. Proposer l'algorithme de la fonction permettant d'obtenir le kième terme de la suite arithmétique : $u_{n+1} = u_n + r$ ($u_0 = 0$) (r est la raison[11] de la suite, $k \in [0, n]$)

[11] la raison est la valeur qui permet de passer d'un terme au suivant

Cinquième partie

Tableaux

10 Tableaux de données de taille fixe

10.1 Exemple introduction en mathématique

Il s'agit de calculer la somme des valeurs d'une liste de valeurs :

1. définir les variables utiles au calcul
   - $A_n \in \mathbb{Z}$
   - $S \in \mathbb{Z}$
2. donner les valeurs de la liste
   - $A = \{2, 6, 4, 5\}$
   - $|A| = 4$ (cardinalité : nombre d'éléments de la liste)
3. calculer s
   - $S = \sum_{i=0}^{3} A_i$

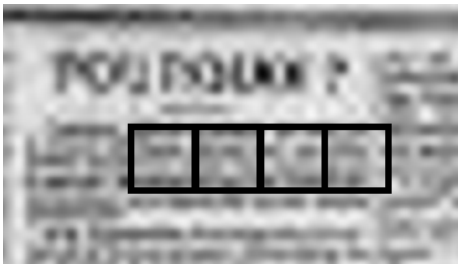
On a donc affaire à :

- une liste finie de valeurs entière : A
- un parcours de la liste, en accédant à chacune de ses valeurs grâce à un indice, pour en calculer la somme (S)

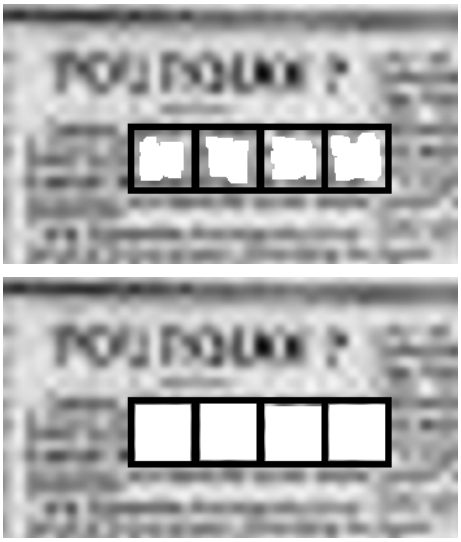
10.2 Exemple : illustration

1. Récupérer une feuille :

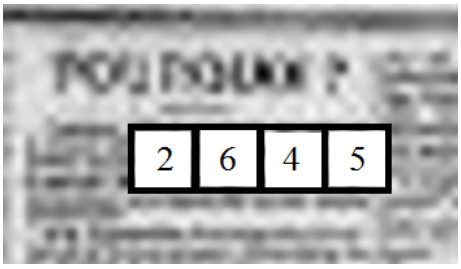
2. Déterminer le nombre de colonnes (ici 1 seule ligne de 4 colonnes) et tracer le tableau :



3. Effacer les cases (elles occupent un espace qui était utilisé auparavant) :



4. Utiliser le tableau pour y mettre les valeurs souhaitées :



10.3 Définition d'un tableau

Tableau

Un *tableau* est une juxtaposition, sous un identifiant unique, d'un nombre fixe de données de même type, auxquelles on accède individuellement grâce à un numéro d'ordre, ou rang, qu'on appelle indice.

Les données qui composent un tableau sont appelées *éléments*. Chaque élément peut contenir une valeur qui peut être utilisée comme n'importe quelle autre donnée.

Algorithme 33 Déclaration de tableaux

```
1: entier tab1[10]          ▷ tab1 est un tableau de 10 entiers
2: reel tab2[24]            ▷ tab2 est un tableau de 24 réels
3: caractere tab3[12]       ▷ tab3 est un tableau de 12 caractères
4: chaine tab4[8]           ▷ tab4 est un tableau de 8 chaines de caractères
```

Attention

On n'accède jamais à la totalité d'un tableau : chaque élément est accessible grâce à un numéro d'ordre, ou rang, ou indice.

En C (et dans de nombreux autres langages) la valeur de l'indice est comprise entre 0 (pour le 1er élément) et taille du tableau -1 (pour le dernier). C'est le principe qu'on

. retiendra ici pour les algorithmes.

La structure de contrôle `for` est généralement bien adaptée au parcours des éléments d'un tableau (voir algo. 10.3 page 95).

Algorithme 34 Accès aux éléments d'un tableaux avec la structure *tant que*

```
1: entier temp[12]                          ▷ temp est un tableau de 12 entiers
2: entier i ← 0                             ▷ i sera la variable d'indice
3: i ← 0                                    ▷ remise à 0 de i
4: pour i variant de 0 à 11 par pas de 1 faire
5:     temp[i] ← 0                          ▷ affecter 0 à la ième élément du tableau
6: fin pour
```

La structure de contrôle `while` est aussi adaptée au parcours des éléments d'un tableau (voir algo. 10.3 page 95).

Algorithme 35 Accès aux éléments d'un tableaux avec la structure *tant que*

```
1: entier temp[12]                          ▷ temp est un tableau de 12 entiers
2: entier i ← 0                             ▷ i sera la variable d'indice
3: i ← 0                                    ▷ remise à 0 de i
4: tant que i < 12 faire                    ▷ Tant que i est inférieur à 12
5:     temp[i] ← 0                          ▷ affecter 0 à la ième élément du tableau
6:     i ← (i + 1)                          ▷ incrémenter i
7: fin tant que
```

Attention

Le contrôle des indices est à la charge du développeur : le compilateur C ne vérifie pas le dépassement des limites.

Dimension

La dimension d'un tableau correspond au nombre d'imbrications de tableaux : chaque élément d'un tableau peut être en effet à son tour un tableau, etc.

10.4 Tableaux à une dimension : vecteurs

Vecteur

On nomme vecteur un tableau ayant une seule dimension. Il nécessite l'utilisation d'une seule variable d'indice pour accéder à chacun de ses éléments.

10.4.1 Déclaration

Syntaxe C 10.18 Déclaration d'un vecteur (tableau 1D)

T idTab[taille] ;
T idTab[] = {liste} ;

où :

- *T* : type de donnée des éléments du tableau,
- *idTab* : identifiant du tableau,
- *taille* : le nombre d'éléments du tableau,
- *liste* : la liste des valeurs données à chacun de ses éléments, séparées par des virgules (la taille sera calculée automatiquement).

Code source 71 – Exemples de tableaux à une dimension en C

```
1 int pop[100]; // population de 100 villes
2 float note[1000]; // notes de 1000 étudiants
3 char voy[6] = {'a','e','i','o','u','y'}; // les 6 voyelles
```

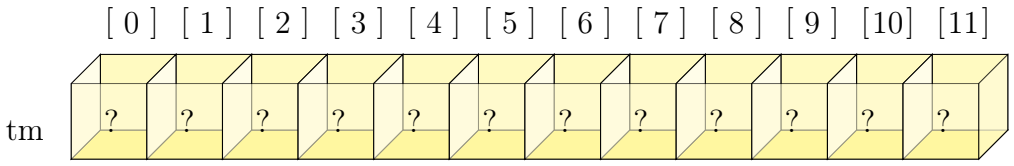
Les 2 premiers tableaux ne sont pas initialisés.

Par exemple, pour déclarer un tableau de 12 mesures ; soit 12 variables du même type et qui portent le même identifiant *temp* :

Code source 72 – Déclaration C d'un tableau de 12 mesures

```
1 int temp[12];
```

La déclaration se traduit par la réservation d'un espace dans la mémoire vive de l'ordinateur (voir Figure ?? en page 112)



10.4.2 Accès aux éléments

Pour accéder aux éléments d'un tableau, on doit indiquer l'indice de l'élément concerné : c'est un nombre entier naturel dans l'intervalle [0,taille-1].

Syntaxe C 10.19 Accès à un élément d'un vecteur (tableau 1D)

idTab[expr]

où :

- *idTab* : identifiant du tableau,
- *expr* : une expression dont la valeur fournit l'indice d'un élément du tableau

Cet élément forme une variable utilisable comme toute autre variable.

Initialisation des tableaux

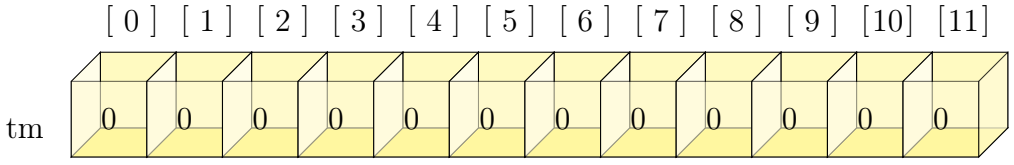
> **Attention**
> Quand un tableau est déclaré (comme toute autre variable), un espace de la mémoire vive lui est alloué : ce dernier a été occupé par d'autres données ou programmes auparavant : il est indispensable d'effacer chaque case du tableau, concrètement en affectant à chacune une valeur unique, généralement 0 pour les tableaux de nombres.

Ainsi pour affecter la valeur 0 chacun des éléments :

Code source 73 – Remmettre à 0 chaque élément d'un tableau C

```
1 int tm[12];
2 int i = 0;
3 for (i = 0; i < 12; ++i) {
4         tm[i] = 0;
5 }
```

Le tableau est alors modifié :



La saisie d'un élément d'un tableau est identique à celle d'une variable élémentaire.

Code source 74 – Saisie d'un élément d'un tableau C

```
1 scanf(fmt, &idTab[indice]);
```

où :

- *fmt* : la chaine de format de la valeur saisie
- *idTab* : l'identifiant du tableau utilisé ; remarquez que l'opérateur & n'a pas été utilisé
- *indice* : une valeur qui donne la position de l'élément choisi (entre 0 et taille du tableau-1)

Par exemple, pour demander la saisie des valeurs des 12 mesures :

Code source 75 – Exemple de saisie des valeurs d'un tableau en C

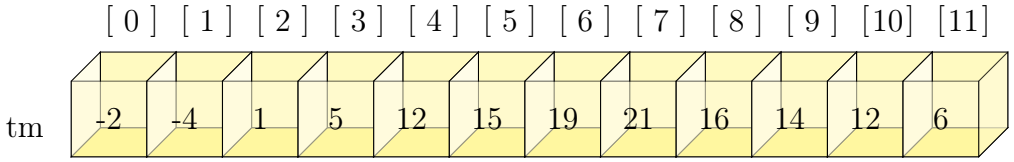
```
1 ...
2 int tm[12];
3 int i = 0;
4 while (i < 12) {
5         scanf("%d",&tm[i]);
6         ++i;
7 }
8 ...
```

Code source 76 – Exemple de saisie des valeurs d'un tableau en C++

```
9  #include <array>
10 ...
11 array <int, 12> temp;
12 int i = 0;
13 while (i < 12) {
14         std::cin >> temp[i];
15         ++i;
16 }
17 ...
```

Le type C++ `array` est un objet contenant un tableau C.

Après la boucle de saisie, les valeurs de chaque élément sont correctement initialisées (voir Figure ?? en page 98)



10.4.3 Taille d'un tableau : opérateur `sizeof`

L'opérateur `sizeof` peut être appliqué à un type de donnée ou à une variable déclarée pour obtenir l'espace mémoire occupé.

Appliquée à un type, il renvoie le nombre d'octets occupés par une variable de ce type.

Code source 77 – Exemple d'utilisation d'un pointeur

```
14   printf("Taille d'un nombre entier : %d octets", sizeof(int));
15   printf("\nTaille de la variable tab : %d octets",sizeof tab);
16   printf("\nNombre d'elements de tab: %d elements\n",
17              sizeof tab / sizeof(int));
```

Résultat d'exécution :

```
Taille d'un nombre entier : 4 octets
Taille de la variable tab : 48 octets
Nombre d'elements de tab: 12 elements
```

10.4.4 Exercices

1. Proposer les algorithmes de fonctions répondant aux questions suivantes :
   - Calculer la moyenne à partir d'un tableau de 4 notes
   - Calculer le nombre de notes supérieures ou égales à la moyenne à partir d'un tableau de 4 notes ; la note moyenne est 10
   - Déterminer la note maximale à partir d'un tableau de 4 notes

- Représenter un histogramme horizontal des valeurs du tableau de 4 notes

2. Proposer le dialogue permettant la saisie et le cumul d'une suite de nombres ; la saisie du nombre 0 arrêtera la répétition, le cumul sera affiché puis l'histogramme

10.5 Tableaux à deux dimensions : matrices

$$\text{Matrice A } a_{i,j} \in A_{m,n} = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{pmatrix}$$

Matrice

On nomme *matrice* un tableau ayant 2 dimensions. On nomme généralement la 1ère dimension *ligne* et la 2ème dimension *colonne* (ou inversement...).

Une matrice nécessite l'utilisation de 2 valeurs d'indices pour accéder à chacun de ses éléments : un indice pour accéder à la ligne, puis le second indice pour accéder à la colonne de cette ligne.

Cela correspond en fait à un tableau à une dimension dans lequel chaque élément est un tableau à une dimension.

10.5.1 Déclaration

Syntaxe C 10.20 Déclaration d'une matrice (tableau 2D)

```
T idTab[taille1][taille2];
T idTab[taille1][] = {{listes}};
```

où :

- *T* est un type de donnée,
- *idTab* est l'identifiant donné au tableau,
- *taille1* est le nombre d'éléments de la 1ère dimension du tableau,
- *taille2* est le nombre d'éléments de la 2ème dimension du tableau,du tableau (peut-être omis si on initialise le tableau lors de la déclaration)
- *listes* est la liste de liste des valeurs entre accolades données à chacun des éléments de la dimension 2, séparées par des virgules.

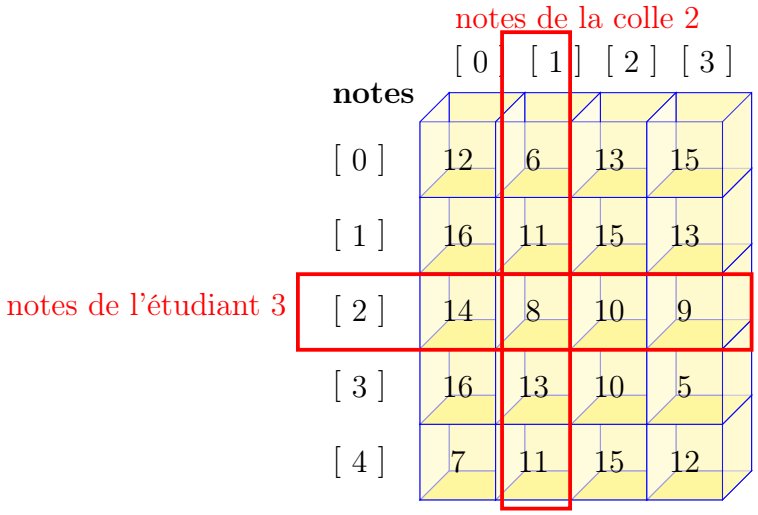
Par exemple, pour déclarer un tableau de 4 notes pour chacun de 5 étudiants :

Code source 78 – Déclaration d'un tableau de 5 x 4 notes

```
1 int notes[5][4];
2 int notes[5][4] = {{12,6,13,15},{16,11,15,13},
3     {14,8,10,9},{16,13,10,5},{7,11,15,12}};
```

La déclaration se traduit par la réservation d'un espace mémoire dans la mémoire vive de l'ordinateur (voir Figure 33 en page 100)

FIGURE 33 – Le tableau occupe 4×5 éléments contigus en mémoire



Un tableau à 5 lignes et 4 colonnes :

- les lignes représentent les notes des étudiants
- les colonnes représentent les notes des colles
- l'intersection d'une ligne et d'une colonne fournit la note d'un étudiant à une colle donnée.

10.5.2 Accès à un élément

Syntaxe C 10.21 Accès à un élément d'une matrice (tableau 2D)

```
idTab[expr1][expr2]
```

où :

- *idTab* : identifiant d'un tableau
- *expr1* : une expression dont la valeur fournit l'indice de la dimension 1 d'un élément du tableau
- *expr2* : une expression dont la valeur fournit l'indice de la dimension 2 d'un élément du tableau

10.5.3 Exercices

1. Proposer le dialogue permettant la saisie des 3 notes de 2 étudiants dans un tableau
2. Proposer les algorithmes de fonctions répondant aux questions suivantes :
   - Calculer et retourner la moyenne générale des 3 notes de 2 étudiants

- Calculer et retourner le nombre d'étudiants ayant leur moyenne supérieure à la moyenne générale ; on considère 2 étudiants et 3 notes pour chacun

10.6 Tableaux multi-dimensionnels (au delà de 2)

Tableau multi-dimensionnel

C'est un tableau dont le nombre de dimensions dépasse 2 (la matrice étant déjà un tableau multi-dimensionnel).

Sa déclaration nécessite la définition du nombre d'éléments de chacun des dimensions.

L'accès à un élément nécessite autant d'indices que de dimensions.

Attention

Attention cependant à l'occupation mémoire de ces tableaux : si la plupart des éléments du tableau ne sont pas utilisés, cette solution n'est surement pas la bonne solution...

11 Chaines de caractères

En C, une chaine peut être représentée par un tableau de caractères :

```
1 char ch[10];
```

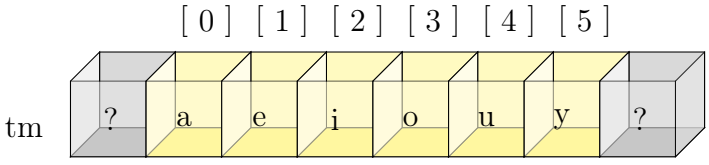
Il existe cependant une différence entre un tableau de caractères utilisé pour mémoriser une suite de lettres (utilisées indépendamment les unes des autres) et un tableau de caractères utilisé pour mémoriser une suite de lettre utilisées comme un tout (un mot, une phrase).

11.1 Tableau de lettres indépendantes

Exemple :

```
1 char v[] = {'a','e','i','o','u','y'};
```

En mémoire le tableau, est représenté ainsi : La première case et la dernière case de cette



représentation correspondent à la fin ou au début d'autres valeurs en mémoire.

Ce tableau comporte 6 éléments : il sera utilisé pour rechercher une lettre, par exemple, mais pas pour afficher en totalité (si on souhaite afficher toutes les voyelles, il faudra les afficher l'une après l'autre)

```c
char v[] = {'a','e','i','o','u','y'};
int i = 0;
for(i = 0; i < (sizeof v / sizeof(char)); ++i) {
        printf("␣%c", v[i]);
}
```

11.2 Chaine de caractère C

Exemple :

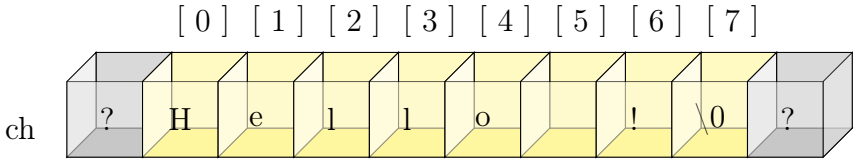
```c
char ch = "Hello␣!";
```

ou bien :

```c
char ch[] = {'H','e','l','l','o','␣','\0'};
```

On remarque qu'un caractère a été ajouté à la fin : il s'agit du marqueur de fin de chaine : '\0'.

En mémoire le tableau, est représenté ainsi :



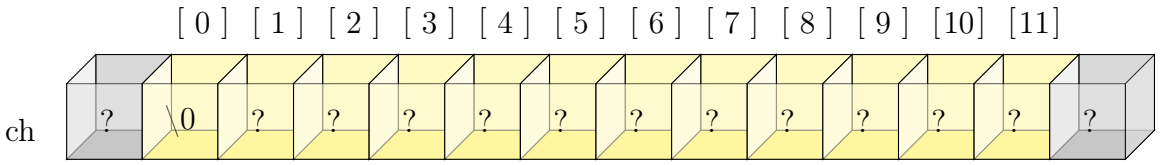
11.2.1 Saisie des chaines de caractères

Lors de l'utilisation d'une chaine de caractères, l'indicateur de fin de chaine est automatiquement ajouté.

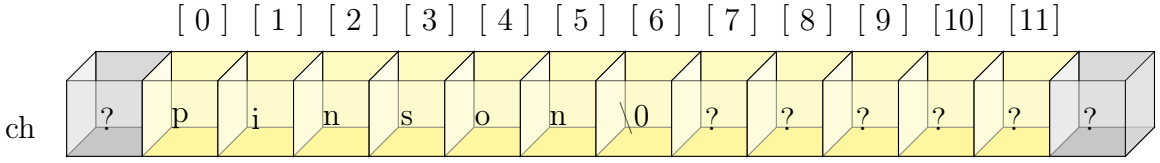
Soit une chaine suffisamment longue pour contenir une saisie :

```c
char ch[12] = "";
```

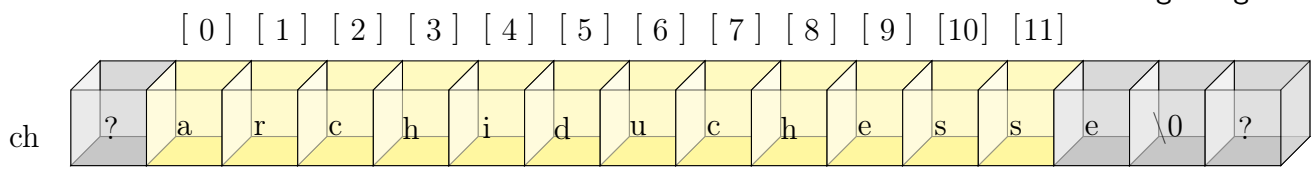
En mémoire, elle contient :



Après saisie du mot 'pinson', la chaine contient :



Après saisie du mot 'archiduchesse', la chaine contient :



La longueur de la valeur saisie étant supérieure à la longueur de la chaine, les données mémoire qui suivent ont été écrasées.

Attention

Lors de la saisie de chaines de caractères, il faut s'assurer que la variable cible aura une capacité suffisante.

Un exemple de saisie contrôlée :

```c
#include <stdio.h>
#define MYBUFFER 256

void vider();
int main()
{
   char ch[MYBUFFER], c;

   printf("Saisir un caractère puis entrée");
   c = getchar();

   vider(); // pour vider le reste du buffer

   printf("Saisir une phrase :");
   fgets(ch, sizeof ch, stdin);

   printf("\nLa phrase est %s, et le caractère est %c\n", ch, c);

   return 0;
}
void vider()
{
   int c;
   while ((c = getchar()) != '\n' && c != EOF)
      ;
}
```

Sixième partie

Structures de données

12 Structure de donnée composée : enregistrement

Les types de données de base - entier, réel, caractères - permettent la représentation de données élémentaires. Lorsqu'on souhaite représenter des objets comportant plusieurs caractéristiques, ils ne sont plus adaptés.

$$\text{un compte} \left\{ \begin{array}{r} \text{numero} \\ \text{titulaire} \\ \text{création} \\ \text{solde} \end{array} \right\} \qquad \text{un titulaire} \left\{ \begin{array}{r} \text{nom} \\ \text{prenom} \\ \text{naissance} \\ \text{adresse} \end{array} \right\} \qquad \text{une adresse} \left\{ \begin{array}{r} \text{numero} \\ \text{rue} \\ \text{complement} \\ \text{codePostal} \\ \text{ville} \end{array} \right\}$$

De manière plus précise, certaines données sont élémentaires (abscisse, ordonnée), d'autres sont composées (point, couleur) :

$$\text{un point} \left\{ \begin{array}{rr} \text{abscisse} & \textit{reel} \\ \text{ordonnee} & \textit{reel} \\ \text{nom} & \textit{caractere} \\ \text{couleur} & \end{array} \right\} \qquad \text{une couleur} \left\{ \begin{array}{rr} \text{rouge} & \textit{entier} \\ \text{vert} & \textit{entier} \\ \text{bleu} & \textit{entier} \end{array} \right\}$$

12.1 L'enregistrement en C

Enregistrement

Un *enregistrement* est une structure de données composée d'un ensemble de variables qui peuvent être de types différents. Un enregistrement constitue un nouveau type utilisable comme tout autre.

12.2 Définition d'un enregistrement

Syntaxe C 12.22 Définition d'un enregistrement (`struct`)

```
struct IdEnreg
{
... membres...
};
```

Syntaxe C 12.23 Définition d'un enregistrement (`struct`) avec alias

```
typedef struct IdEnreg
{
... membres...
} Alias;
```

où :

- `typedef` : mot réservé introduisant la définition d'un alias
- `struct` : mot réservé introduisant la définition d'un enregistrement
- *IdEnreg* : identifiant attribué à l'enregistrement (la 1ère lettre de l'identifiant sera en majuscules selon notre convention)
- *membres* : liste des déclarations des variables composant cet enregistrement
- *alias* : alias donné à la définition de l'enregistrement

Remarquer le ';' en fin de déclaration.

- la notion de 'couleur' peut être décrite par l'enregistrement suivant :

Code source 79 – Définition de l'enregistrement Couleur

```
1 struct Couleur
2 {
3   int r, // composante red
4     g, // composante green
5     b; // composante blue
6 };
7 typedef struct Couleur Couleur;
```

- la notion de point dans le plan peut être décrite par son abscisse et son ordonnée et sa couleur :

Code source 80 – Définition de l'enregistrement Point

```
1 struct Point
2 {
3   float x, // abscisse
4         y; // ordonnée
5   Couleur c; // couleur du point
6 };
7 typedef struct Point Point;
```

Depuis la version **C99**, il est possible d'initialiser les membres de la structure à la déclaration.

12.3 Déclaration d'une variable de type enregistrement

Le nouveau type de données ainsi créé peut être utilisé pour déclarer des variables :

Code source 81 – Déclaration de 2 couleurs

```
1 Couleur c1,
2     c2 = {255,255,255};
```

La variable c1 n'est pas initialisée ; c2 est initialisée à blanc (composants de couleur à 255).

Code source 82 – Déclaration d'un point

```
1 Point p = {0,0,{0,0,0}};
```

Le point p est initialisé : c'est le point à l'origine, il a le noir comme couleur (composants de couleurs à 0).

12.4 Déclaration d'une constante de type enregistrement

Le nouveau type de données ainsi créé peut être utilisé pour déclarer des variables :

Code source 83 – Déclaration de 2 constantes couleurs

```
1 const Couleur BLANC = {255,255,255},
2         NOIR = {0,0,0};
```

Ces valeurs peuvent utilisées pour initialiser des variables de même type :

Code source 84 – Déclaration de 2 couleurs initialisées

```
1 Couleur c1 = BLANC,
2     c2 = NOIR;
```

Le point p est initialisé : c'est le point à l'origine, il a le noir comme couleur.

12.5 Accès aux membres d'une donnée enregistrement

Les membres d'une donnée enregistrement sont accessibles en utilisant une notation pointée : l'identifiant de la variable structure, un point, l'identifiant de la variable membre :

Syntaxe C 12.24 Accès au membre d'un enregistrement

id.membre

où :

- *id* : identifiant d'un objet (variable ou constante) de type enregistrement
- *membre* : nom d'un des membres de l'enregistrement

Exemple :

Code source 85 – Saisir les caractéristiques d'un point

```
1  Point p;
2  // saisie des coordonnées de p
3  scanf("%f %f", &p.x, &p.y);
4  // saisie des composants de la couleur de p
5  scanf("%d %d %d", &p.c.r, &p.c.v, &p.c.b);
```

12.6 Tableaux d'enregistrements

Comme tout autre type de données, l'enregistrement peut être utilisé dans des tableaux :

Syntaxe C 12.25 Déclaration d'un tableau d'enregistrement en C

```
struct IdEnreg idTab[NB];
/* ou bien, si un alias a été défini :*/
IdEnreg idTab[NB];
```

où :

- `struct` *IdEnreg* : identification de l'enregistrement (type du tableau)
- *NB* : nombre d'éléments du tableau (sa taille)
- *idTab* : identifiant du tableau

Code source 86 – Saisie d'un tableau de points

```
28   Point p[NB];
29   int i = 0;
30   for(i=0; i < NB; i=i+1) {
31     /* saisie des coordonnées de p */
32     scanf("%f %f", &p[i].x, &p[i].y);
33     /* saisie des composants de la couleur de p */
34     scanf("%d %d %d", &p[i].c.r, &p[i].c.g, &p[i].c.b);
35   }
```

12.7 Exercices

On souhaite créer une application utilisant des molécules dont voici quelques exemples :

- l'eau : H_2O
- le méthane : CH_4
- l'acide sulfurique : H_2SO_4
- le glucose : $C_6H_{12}O_6$

- le benzène : C_6H_6

Proposer la définition d'un enregistrement permettant de représenter une molécule, dans laquelle on se limitera à un maximum de dix atomes.

Septième partie

Sous-programmes

13 Sous-programmes

Un sous-programmes est un moyen de nommer une action complexe, généralement composée de plusieurs instructions ou calculs élémentaires,

- soit parce qu'on sera amené à l'utiliser plusieurs fois dans un ou plusieurs autres algorithmes (réutilisation),
- soit de manière à rendre la structure d'un algorithme plus claire et lisible (issue de la décomposition d'un algorithme complexe en sous-algorithmes)

Il est généralement issu de la décomposition d'un problèmes en sous-problèmes : le mode de résolution des sous-problèmes aboutit généralement à la définition d'algorithmes qui vont composer le mode de résolution global du problème.

Sous-programme

Un *sous-programme* est une forme particulière d'algorithme : on ne l'exécutera jamais directement, il sera *toujours appelé par un autre algorithme*. Un sous-programme attendra généralement des données en entrée qui lui seront fournies par l'algorithme qui l'appelle : ce sont les paramètres qui assurent ce mode d'échange ; à la fin de son exécution, il pourra fournir en retour un résultat.

Nous distingueront 2 formes de sous-programmes :

- les *fonctions*, ou sous-programme *expression* : sous-programme qui *retourne une valeur* d'un certain type à l'issue de son exécution ; on peut ainsi directement utiliser une fonction comme une valeur dans une expression
- les *procédures*, ou sous-programme *action* : sous-programmes qui exécutent des instructions et ne renvoie pas de valeur directement ; on ne peut que les appeler comme une instruction sans attendre de valeur de retour.

Modularité

La *modularité* est la qualité d'un programme à être divisé en sous-programmes ou modules, éventuellement réutilisables.

13.1 Appel d'un sous-programme, arguments, paramètres

Lorsqu'on appelle un sous-programme, il suffit de le nommer et d'indiquer, généralement entre parenthèses, les valeurs qui sont échangées entre l'appelant et le sous-programme appelé (paramètres).

Paramètres

Les paramètres correspondent aux informations échangées lors de l'appel d'un sous-programme.

Paramètres formels

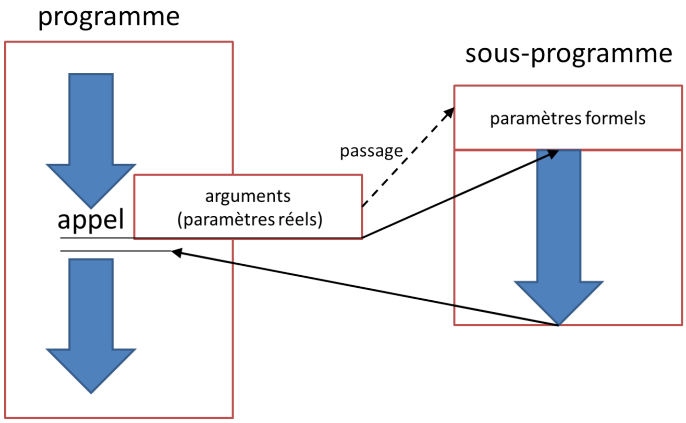
Les *paramètres formels* définissent les valeurs attendues par le sous-programme pour assurer son traitement.

Arguments, paramètres réels, paramètres effectifs

Les *arguments*, ou *paramètres réels* ou *paramètres effectifs* sont les valeurs effectivement passées lors de l'appel du sous-programme par l'appelant , et copiées dans les paramètres formels généralement .

Lors de l'appel d'un sous-programme, les arguments sont passés aux paramètres formels selon leur position : le 1er argument est passé au 1er paramètre, le 2ème argument au 2ème paramètre, etc.

FIGURE 34 – Appel d'un sous-programme, passage des arguments et retour



13.2 Modes de passage des paramètres

Modes de passage des paramètres

On distingue 3 modes de passage des paramètres entre l'appelant et l'appelé (le sous-programme invoqué) :

- le mode de passage *par valeur*, en lecture seule : le paramètre formel du sous-programme est une variable locale qui est initialisée à chaque appel par la valeur de l'argument passé par l'appelant ;
- le mode de passage *comme résultat*, en écriture seule : le paramètre formel du sous-programme est une variable locale dont la valeur sera copiée dans l'argument après chaque appel.
- le mode de passage *en lecture-écriture* : le paramètre formel du sous-programme

est une variable locale qui est initialisée à chaque appel par la valeur de l'argument et dont la valeur après chaque appel est recopiée dans l'argument, après qu'elle ait été modifiée

En C/C++, les modes de passage sont réduits à 2 :

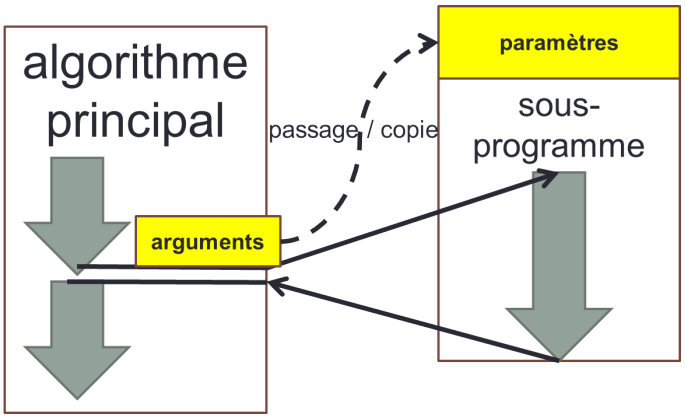
- le mode de passage *par copie ou par valeur* : la valeur de l'argument est copiée vers le paramètre formel ; ce mode de passage équivaut au mode en lecture
- le mode de passage par *adresse/pointeur* : l'adresse de l'argument est copiée vers le paramètre formel qui est défini comme pointeur ; ce mode de passage équivaut généralement au mode en lecture/écriture, mais peut être contraint en lecture seule (en qualifiant la variable de `const`).[12]

En C, les paramètres sont passés par défaut :

- par copie, en lecture seule, pour les variables ayant un type élémentaire ou un type enregistrement
- par adresse/pointeur, en lecture/écriture, pour les variables de type tableau

Le compilateur vérifie que le nombre d'arguments correspond au nombre de paramètres et que les valeurs copiées sont bien compatibles (mêmes types de donnée).

FIGURE 35 – Passage par valeur à un sous-programme



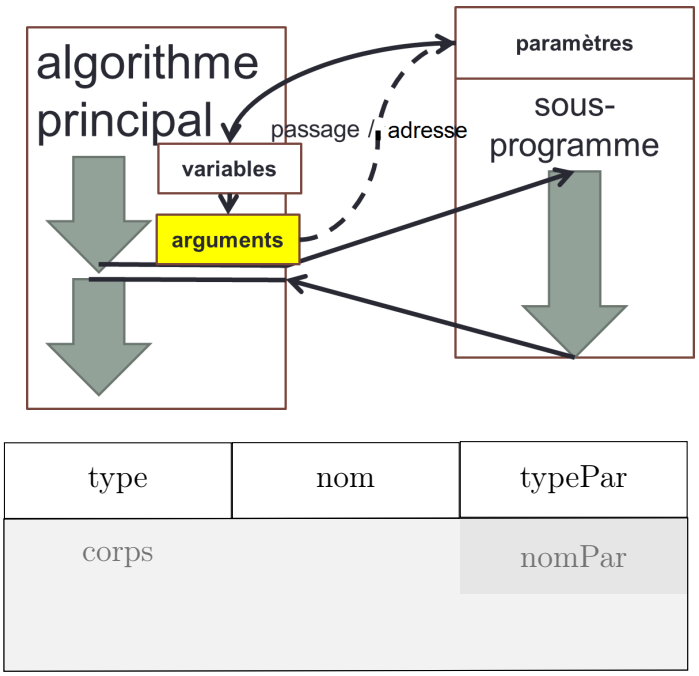
13.3 Corps de la fonction : privé

D'une fonction, seul le prototype est connu des autres fonctions : il constitue l'interface, le moyen de communiquer avec les autres fonctions, il est public, partagé

Les identifiants des paramètres formels et le corps de la fonction sont, quant à eux, privés, locaux à la fonction.

[12] Le C++ propose le mode de passage *par référence* pour lequel le paramètre formel est un alias de l'argument : ce mode de passage équivaut généralement au mode en lecture/écriture sauf s'il s'agit d'une référence constante.

FIGURE 36 – Passage par adresse/pointeur à un sous-programme



14 Fonctions, sous-programmes "expression"

14.0.1 Analogie entre fonctions mathématiques et fonctions informatiques

Un algorithme est également assimilable à une fonction qui, à des données initiales, associe le résultat de leur transformation, avec cependant une différence majeure.

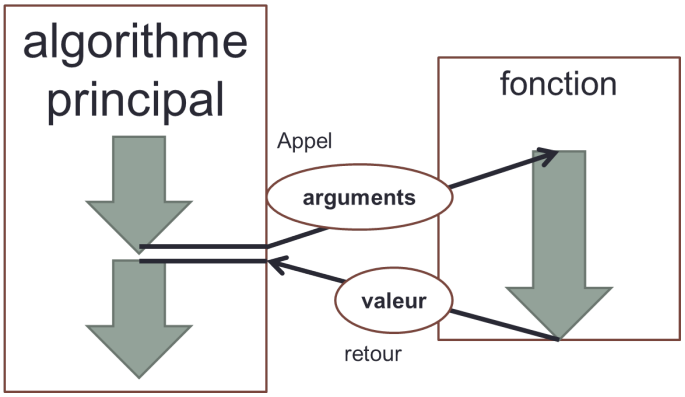
une *définition formelle* en mathématique : Une fonction mathématique met en relation 2 grandeurs de manière à ce que la connaissance de la première permet de connaitre la deuxième : $f : X \to Y$, $y \in Y$, et $x \in X$, $f(x) = 2x + 3$; la connaissance de x me permet de connaitre y.

une *définition procédurale* en informatique : L'algorithme informatique va, quant à lui, décrire les étapes du mode de calcul de la somme.

Algorithme 36 Calcul de $f(x) = 2x + 3$

```
1: Fonction ENTIER F((entier x))          ▷ entier : domaine de définition
2:     entier r ← 0                       ▷ s : variable de calcul
3:     r ← (2 × x + 3)
4:     return r                           ▷ la valeur de f est celle de r
5: fin Fonction
```

14.1 Déclaration d'une fonction : prototype ou signature



Syntaxe C 14.26 Déclaration d'une fonction

T idFonc(declParam) ;

où :

- `T` : type de valeur retourné par la fonction
- `idFonc` : identifiant de la fonction
- `declParam` : déclaration individuelle de chacun des paramètres (peut être vide) ; seuls les types de chacun des paramètres sont indispensable dans la déclaration d'une fonction.

Prototype

Le *prototype* d'une fonction correspond à son entête, et contient :

- son type
- son identifiant
- les types de ses paramètres

La *signature*, d'une fonction correspond à son entête, hormis son type , et contient :

- son identifiant
- les types de ses paramètres

Code source 87 – Exemple de prototype de la fonction doubler et C

```
1 int doubler(int, int);
```

14.2 Définition d'une fonction

La définition d'une fonction reprend le prototype et détaille le bloc formant le corps de la fonction.

Syntaxe C 14.27 Définition d'une fonction

```
T idFonc(declParam)
{
    ... corps ...
    return expr ;
}
```

où :

- `T` : type de valeur retourné par la fonction ; obligatoire depuis **C99**(le type `int` est pris par défaut dans les versions antérieures)
- `idFonc` : identifiant de la fonction
- `declParam` : déclaration individuelle de chacun des paramètres (peut être vide)
- `return`, obligatoire : instruction de retour de la valeur ; cette instruction quitte immédiatement la fonction et revient au point d'appel
- `expr` : expression du même type que T dont la valeur est retournée

Conseil

Une *fonction* ne devrait comporter que des instructions de calcul et des instructions de contrôle. Toute autre instruction (entrées/sorties, par exemple), introduisant une interruption de l'exécution et risquant de provoquer des effets de bord, devraient être évitée.

Attention

Le type de retour d'une fonction peut être n'importe lequel des types de base ou des enregistrements, mais il ne peut être un tableau.

Si on doit retourner un tableau dans une fonction, il sera nécessaire de passer par une allocation dynamique et le retour du pointeur correspondant (cf. cours année 2)

14.3 Pré-condition

pré-condition

La *pré-condition* d'une fonction est la condition, qui appliquée aux valeurs des arguments reçus, permet de garantir sa bonne exécution et le retour d'une valeur cohérente suite à cette exécution.

L'appelant d'une fonction *doit s'assurer* que les arguments passés à la fonction lors de l'appel respectent cette pré-condition.

La pré-condition représente une forme de contrat de bonne exécution.

En C, la pré-condition pourra être traduite par une instruction `assert` (fichier d'entête assert.h). Dans le cas où l'assertion échoue, le programme est arrêté immédiatement .

Attention
L'assertion en C n'est active qu'en mode DEBUG. Voir la section 26 page 156

14.4 Exemple

Code source 88 – Exemple de déclaration de paramètres formels en C

```
1 int tripler(int v)
2 {
3  return (3*v);
4 }
```

ou :

- en ligne 1 : `v` est un paramètre formel, il contiendra la valeur à utiliser dans la fonction

Les identifiants des paramètres formels ne sont connus que dans la fonction où ils sont définis. Ils forment une variable locale à la fonction.

Exemple :

Code source 89 – Exemple d'appel d'une fonction en C

```
1 int a = 0,
2   b = 0;
3 a = tripler(2);
4 b = tripler(a);
```

ou :

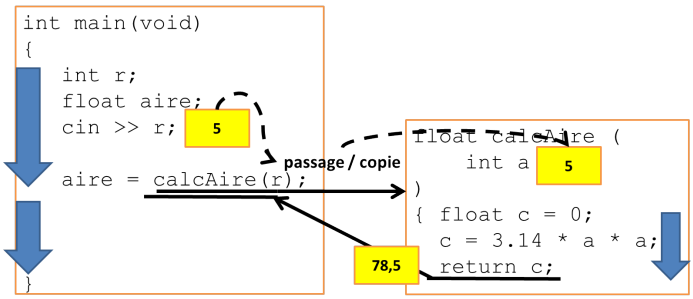
- en ligne 3 : `2` est la valeur de l'argument passée à la fonction
- en ligne 4 : la valeur de `a` est la valeur de l'argument passée à la fonction

Le paramètre `v` de la fonction va prendre comme valeur `2`, puis la valeur de `a`.

14.5 Modes de passage des paramètres

Conseil
Les paramètres d'une fonction devraient être passés en lecture seule afin d'éviter les effets de bords, modifications d'éléments extérieure à la fonction.

FIGURE 37 – Passage par valeur à une fonction



14.6 Quitter et retourner une valeur : `return`

L'instruction `return` quitte la fonction en retournant une valeur. La suite de l'exécution reprend à l'endroit de l'appel de la fonction, par substitution de la valeur retournée à cette de la fonction

Syntaxe C 14.28 Instruction `return`

return expr ;

où :

- `return` : l'instruction de retour
- *expr* : valeur renvoyée au point d'appel, celle-ci étant du même type que la fonction

Code source 90 – Exemple de fonction f et appel de la fonction f en C

```
1 int f(int n)
2 {
3         return (n*2);
4 }
5 int main(void)
6 {
7         f(2);
8 }
```

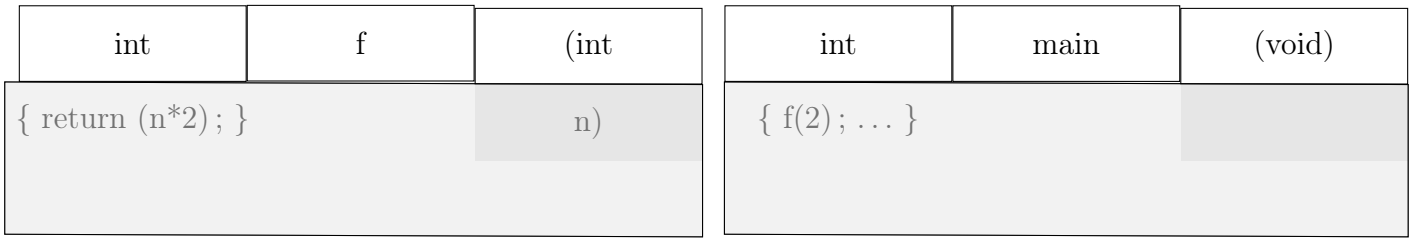


FIGURE 38 – Seuls les prototypes sont visibles, les corps sont inaccessibles

14.7 Exercices

1. *somme* attend 2 nombres entiers et retourne la somme des 2 nombres
2. *abs* attend 1 nombre entier et retourne la valeur absolue du nombre
3. *max2* attend 2 nombres entiers et retourne le plus grand des 2
4. *max3* attend 3 nombres entiers et retourne le plus grand des 3 (utiliser la fonction précédente...)
5. *carre* attend 1 nombre entier et retourne le nombre élevé au carré
6. *puissance* attend 2 nombres entiers et retourne le premier nombre élevé à la puissance du second
7. *deg2Fahr*, qui convertit une température exprimée en Celsius en une température exprimée en Fahrenheitattend un nombre réel retourne la conversion en degré Fahrenheit du nombre (celsius = 5 * (fahrenheit ? 32) / 9
8. *fahr2Deg*, la fonction inverse, qui convertit une température exprimée en Fahrenheit en une température exprimée en Celsius
9. *fact* attend 1 nombre entier et retourne la factorielle de ce nombre (rappel : n ! = n * n-1 * n -2 * ... * 2 * 1)
10. *estPair* attend 1 nombre entier, et retourne un booléen à VRAI si le nombre est pair, à FAUX dans le cas contraire
11. *arrondi* attend 1 nombre réel et retourne le nombre entier arrondi le plus proche (+12,4 donne +12, +12,5 donne +13, -12,4 donne -12, et -12,5 donne -13).
12. *position* attend 1 tableau de NB nombres entiers (un vecteur) et un nombre entier et retourne l'indice du nombre trouvé ou -1 sur le nombre n'a pas été trouvé
13. *tabCompte1* attend 1 tableau de NB nombres entiers (un vecteur) et retourne la somme des nombres du tableau
14. *tabCompte2* attend 1 tableau de NB nombres entiers (un vecteur), un indice début et un indice fin et retourne la somme des nombres du tableau entre les indices début et fin
15. *sommeDiag* attend 1 tableau d ?entiers à 2 dimensions, NBXNB, (une matrice carrée), et retourne la somme des nombres de la diagonale de la matrice
16. *compteVoyelles* attend 1 tableau de NB caractères et retourne le nombre de voyelles du tableau
17. *estPalindrome* attend 1 tableau de NB caractères (un mot) et déterminer si le mot est un palindrome ou pas [13]
18. *f1* est définie par : $f1(x) = \begin{cases} 3x^2 + x + 1 & \text{si x >= 1} \\ 0 & \text{sinon} \end{cases}$ Elle attend un nombre entier et retourne la valeur calculée

[13] palindrome : mot qui peut être lu de gauche à droite et de droite à gauche (gag, elle, etc.)

19. *f2* est définie ainsi : $f2(x) = f1(x) + f1(x-1) + ... + f1(1)$: Elle attend un nombre entier et retourne la valeur calculée

20. Proposer une fonction qui n'attend aucun paramètre et qui renvoie un point 2D à l'origine (un point est un enregistrement)

21. Proposer une fonction qui attend les coefficients d'un polynôme du 2nd degré et qui retourne les racines (nombre de racines, et valeurs des racines si applicable)

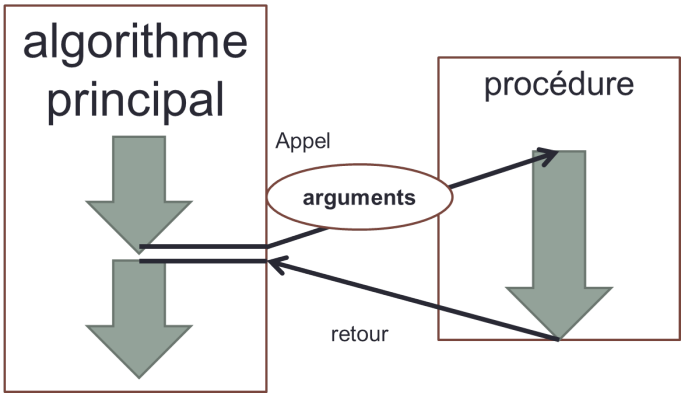
15 Procédures, sous-programmes "action"

Procédure

Une *procédure* est un sous-programme qui exécute un certain nombre d'instructions sans fournir de valeur de retour après son exécution. C'est un sous-programme de type 'instruction' ou 'action' (non utilisable dans une expression, contrairement aux fonctions).

Attention

La terminologie du langage C ne cite que le terme 'fonction' et ne distingue pas une véritable fonction d'une procédure.



15.1 Déclaration et définition d'une procédure

La déclaration d'une procédure définit simplement le prototype de la procédure.

La définition d'une procédure définit l'ensemble de la procédure : prototype et corps.

Syntaxe C 15.29 Déclaration d'une procédure

```
void idProc (declParam) ;
```

Syntaxe C 15.30 Définition d'une procédure

```
void idFonc (declParam)
{
    ... corps ...
    return ;
}
```

où :

- `void` : type d'une procédure (dite *fonction void*)
- *idProc* : identifiant de la procédure
- *declParam* : déclaration individuelle des paramètres de la procédure (la liste peut être vide)
- *corps* : instructions composant la procédure ;
- `return`, optionnel : quitte immédiatement la fonction et revient au point d'appel ; en cas d'absence, le retour en implicite en fin de procédure.

15.1.1 Exemple 1

:

Code source 91 – Déclaration et définition de la fonction void 'afficherMenu'

```
1  void afficherMenu();    // déclaration
2
3  int main()
4  {
5   ... (cf. exemple suivant)
6  }
7
8  void afficherMenu() {    // définition
9    printf("Menu\n=====
10    "Choix 1 : somme de 2 nombres\n"
11    "Choix 2 : produit de 2 nombres\n"
12    "Choix 3 : quotient de 2 nombres\n"
13    "... d'autres choix ...\n"
14    "Choix 0 : quitter le programme\n"
15    "Votre choix ?");
16 }
```

La fonction 'afficherMenu' ne définit aucun paramètre.

Code source 92 – Appel de la fonction 'afficherMenu'

```
1  ...
2  int main()
```

```
3  {
4    /* DECLARATIONS */
5    int choix = 0;
6    /* TRAITEMENT */
7      /* demander la choix et répéter
8         tant que le choix est différent de 0 */
9    afficherMenu();  // appel de la fonction 'afficherMenu'
10   scanf("%d", &choix);
11   while (choix != 0) {
12     /* selon le choix de l'utilisateur,
13        effectuer le traitement approprié */
14     ...
15     afficherMenu(); // appel de 'afficherMenu'
16     scanf("%d", &choix);
17    } ;
18 }
19 ...
```

15.1.2 Exemple 2

Code source 93 – Déclaration et définition de la fonction void 'afficherTableau'

```
1  /* Exemple de définition de type tableau */
2  #define NB 12
3
4  void afficherTableau(char *, int [NB]);
5
6  int main()
7  {
8  ... (cf. exemple suivant)
9  }
10
11 void afficherTableau(char * titre, int t[NB])
12 {
13  int i = 0;
14  printf("␣%s\n",titre);
15  for (i=0;i<NB;i++) {
16    print(f"␣%d", t[i]);
17  }
18  printf("\n");
19 }
```

Cette fonction définit 2 paramètres : une chaine de caractères et un tableau

Code source 94 – Appel de la fonction 'afficherTableau'

```
1  int main()
2  {
3    /* DECLARATIONS */
4    Tableau t1, t2, t3;
```

```
5    int i;
6    /* INITIALISATION */
7      /* demander la saisie de t1, t2 et t3 */
8    ...
9    /* RESULTAT */
10     /* affichage de t1 */
11   afficherTableau("Tableau1",t1);
12     /* affichage de t2 */
13   afficherTableau("Tableau2",t2);
14     /* affichage de t3 */
15   afficherTableau("Tableau3",t3);
16
17  return 0;
18 }
```

Les fonctions bien nommées peuvent se substituer à la documentation du programme.

15.2 Modes de passage des paramètres

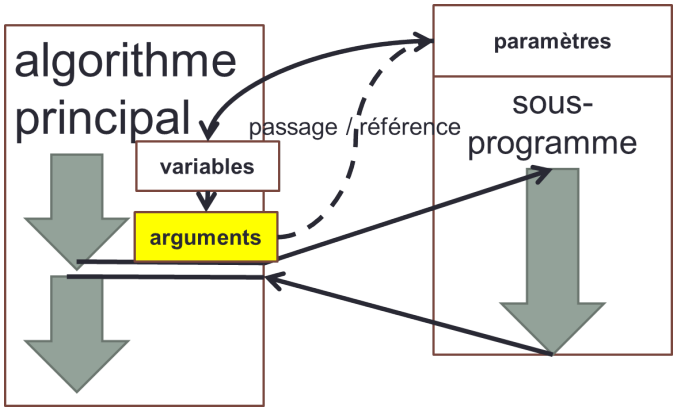
Plusieurs modes de passage des paramètres sont proposés pour les procédures :

- passage par copie de valeur (identique aux fonctions)
- passage par adresse (pointeur) (non approfondi cette année) ; par défaut, les tableaux sont passés par adresse

15.2.1 Passage par pointeur

Les arguments passés par adresse au sous-programme sont des variables, pas des valeurs littérales.

De plus, ce n'est pas la valeur de la variable qui est passée au sous-programme, mais son adresse : le sous-programme aura ainsi un accès direct au contenu de la variable par l'intermédiaire de son paramètre formel (un pointeur).



On indique qu'un sous-programme reçoit une adresse en définissant le paramètre formel comme pointeur (en complétant le type par '*'). L'appelant devra alors passer l'adresse d'une variable en utilisant l'opérateur d'adressage '&'.

15.2.2 Exemple 1

Code source 95 – Déclaration et définition de la procédure 'saisirUnNombre'

```
1  void saisirUnNombre(int * , int );
2
3  int main()
4  {
5  ... (cf. exemple suivant)
6  }
7  void saisirUnNombre(int * n, int valeurMini)
8  {
9   int nombre = 0;
10  /* demander la saisie controlée d'un nombre */
11  scanf("␣%d", &nombre);
12  while (nombre <= valeurMini) {
13   printf("erreur␣:␣recommencer␣:");
14   scanf("␣%d", &nombre);
15  }
16  /* actualiser la valeur du paramètre */
17  *n = nombre;
18 }
```

Remarquer la déclaration des paramètres formels comportant l'opérateur de référencement '*' (le paramètre formel est un pointeur : il contiendra l'adresse de l'argument passé au moment de l'exécution)

Code source 96 – Appel de la procédure 'saisirUnNombre' :

```
1  ...
2  int main()
3  {
4          int n1, n2;
5          saisirUnNombre(&n1, 0);
6          saisirUnNombre(&n2, n1);
7          ...// suite du traitement
8  }
9  ...
```

- Dans le 1er appel, l'adresse de *n1* est passée en paramètre : la saisie modifie directement *n1*.
- Dans le 2eme appel, l'adresse de *n2* est passée en paramètre : la saisie modifie directement *n2* ; la valeur de *n1* est passé par copie/valeur.

15.2.3 Comparaison modes de passage par valeur / par adresse

Code source 97 – passage par valeur

```
18 void afficher(int a)
19 {
20   printf("␣%d",a);
21 }
22 int main()
23 {
24   int n = 6;
25   afficher(n);
26 }
```

Code source 98 – passage par pointeur

```
27 void doubler(int * a)
28 {
29   /* le contenu pointé par a est doublé */
30   *a = *a * 2;
31 }
32 int main()
33 {
34   int n = 6;
35   doubler(&n);
36 }
```

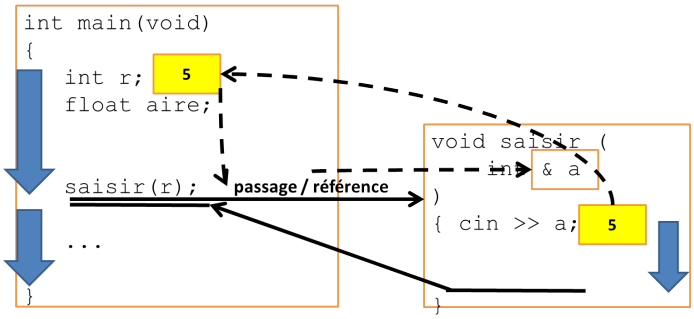
Passage par adresse

Dans le mode de passage par adresse, l'adresse de l'argument est copiée dans le paramètre formel, un pointeur : toute modification de la valeur du paramètre formel est donc directement appliquée au paramètre réel.

15.2.4 Passage par référence en C++

Le C++ propose un mode de passage par référence : le paramètre formel devient un alias de l'argument ; le sous-programme a donc un accès direct à la valeur de l'argument.

FIGURE 39 – Passage par référence à un sous-programme en C++



15.3 Quitter normalement : `return`

L'instruction `return`, optionnelle, quitte la procédure immédiatement sans retourner de valeur. La suite de l'exécution reprend à l'instruction qui suit l'appel de la procédure.

Syntaxe C 15.31 Instruction `return`

```
return ;
return void ;
```

où :

- `return` : instruction de retour à l'instruction qui suit le point d'appel
- `void` : type nul

15.4 Quitter brutalement : `exit`

La fonction `exit` stoppe brutalement l'exécution du programme dans sa totalité. Le contrôle est rendu à la tâche qui avait démarré le programme.

Cette fonction doit n'être utilisé que dans les cas d'erreurs impossible à gérer.

Syntaxe C 15.32 Instruction `exit`

exit(exprNum) ;

où :

- `exit` : identifiant de la fonction
- *exprNum* : expression numérique qui sera retournée à la tâche qui a lancé le programme ; cette dernière pourra ainsi décider de la marche à suivre en fonction de la valeur.

15.5 Exercices

Pour chacune des procédures, décrire son algorithme et donner un exemple de son appel dans un algorithme principal :

- *operation* attend 2 nombres entiers a et b, un caractère c, et écrit "a c b vaut r" où r est le résultat de l'opération : + pour l'addition, - pour la soustraction, * pour la multiplication et / pour la division (dans les autres cas, on affichera "opération inconnue").
- *afficherEtudiant* attend 1 variable de type structure étudiant (numéro : entier, nom : chaine de caractères) et écrit "l'étudiant de numéro X se nomme Y", où X et Y correspondent au numéro et au nom de l'étudiant.
- *afficherTab1* e. attend 1 tableau de NB entiers (vecteur) et un caractère c f. et écrit les valeurs du tableau : horizontalement sur une ligne si le caractère c vaut H ou verticalement sur une colonne dans le cas contraire.
- *AfficherTab2* attend 1 tableau d'entiers à 2 dimensions, NBLIG X NBCOL (matrice), et écrit le contenu du tableau sous forme d'une matrice (NBLIG lignes et NBCOL colonnes).
- *AfficherDiagonale* attend 1 tableau d'entiers à 2 dimensions NB X NB (matrice carrée) et écrit les valeurs de la diagonale de la matrice.
- Question : est-il possible de transformer la procédure *operation* en une fonction qui renverrait le résultat du calcul ?
  - Proposer l'algorithme d'une fonction et son appel dans un algorithme principal

- Critiquer cette solution
- Quelle est la spécificité d'un sous-programme *fonction* au regard d'un sous-programme *procédure*

Huitième partie
Récursivité

16 Récursivité

La récursivité, en informatique, est une démarche de résolution de problèmes qui définit au moins 2 cas :

- un (ou plusieurs) cas de résolution terminal, sans appel récursif
- les autres cas de résolution en appliquant le même algorithme avec un jeu de données réduit qui fournit une solution intermédiaire.

Un algorithme récursif s'appuie sur une définition par récurrence d'une suite de valeurs à utiliser. Il s'appelle généralement lui-même en passant en paramètres les données permettant les calculs intermédiaires.

Une fonction récursive comporte toujours au moins une alternative :

Algorithme 37 La récursivité

```
1: si conditionDArret alors
2:     cas trivial
3: sinon
4:     cas général
5: fin si
```

qui permet d'orienter vers au moins 2 cas :

- le *cas trivial* : ce cas ne doit pas effectuer d'appel récursif ; il est déterminé par la condition d'arrêt de l'appel récursif
- le *cas général* : ce cas effectue un appel récursif avec diminution de la valeur de son paramètre (= traitement d'un ensemble moins grand doit converger vers le cas d'arrêt d'appel récursif)

16.0.1 Récursivité simple

Un exemple classique est celui du calcul de la factorielle d'un nombre. La factorielle peut être traitée efficacement en utilisant un algorithme itératif, mais plus élégamment en utilisant la récursivité.

La factorielle d'un nombre n peut être définie ainsi :

$n \in \mathbb{N}$

$$factorielle(n) = \begin{cases} 1 & \text{si } n = 0 \\ n * factorielle(n-1) & \text{sinon} \end{cases}$$

L'algorithme correspondant sera :

Algorithme 38 Calcul de la factorielle d'un nombre - récursivité

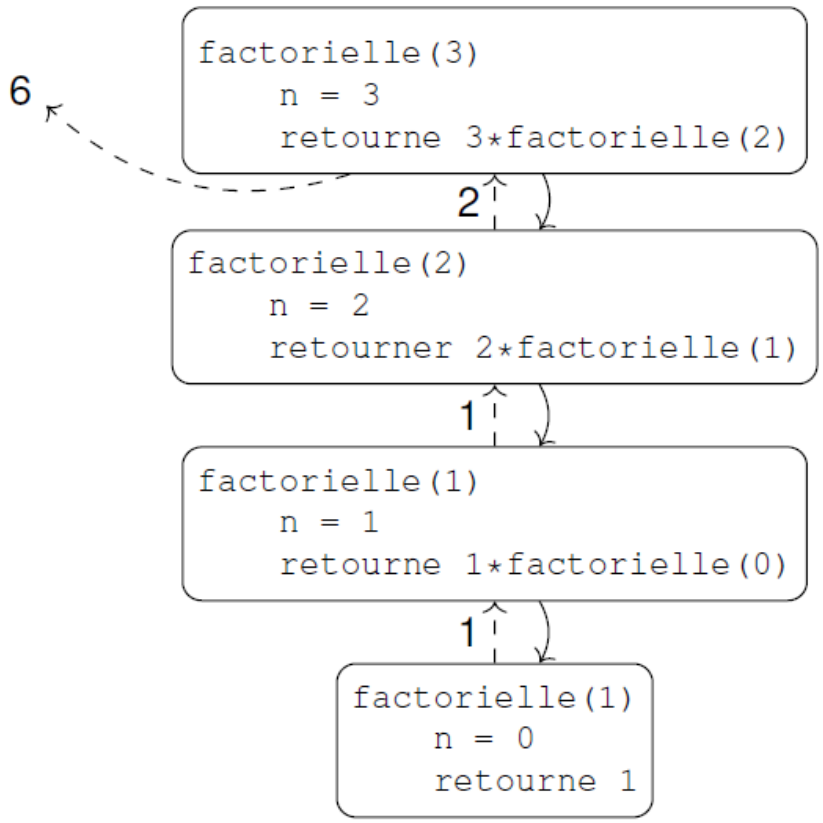
1: **Fonction** FACTORIELLE(n : entier) : entier
Pre-condition: ($n >= 0$)
2: **si** $n = 0$ **alors**
3: retourner 1
4: **sinon**
5: retourner n * factorielle(n-1)
6: **fin si**
7: **fin Fonction**

Code source 99 – Fonction de calcul récursif de la factorielle d'un nombre

```
1 /* pre-condition : (n >= 0)*/
2 long factorielle(long n)
3 {
4         if (n == 0) {return 1;}
5         else {return (n*factorielle(n - 1));}
6 }
```

La fonction récursive met en attente chaque appel récursif jusqu'à ce que la valeur de factorielle(n - 1) ait été calculée et retournée : on a ainsi un arbre d'appel qui va devoir empiler des calculs intermédiaires dans l'attente du résultat renvoyé par l'appel de factorielle : $n * factorielle(n-1)$

Par exemple, dans le cas de la fonction factorielle, le développement du calcul de factorielle(3) serait le suivant :



Pour une grande valeur de n, l'arbre d'appel sera de grande taille : cela aura une incidence non négligeable sur les performances (temps et espace de données réservé pour la mémorisation des résultats intermédiaires)[14].

16.0.2 Notion d'ordre dans les appels récursifs

Ordre

La notion d'ordre définit le nombre d'appels réalisés dans une instruction d'appel récursif. L'ordre est généralement 1 pour signifier que la fonction récursive est appelée une seule fois à chaque appel

Par exemple, le calcul du nième terme de la suite de Fibonacci utilise un appel récursif d'ordre 2 :

$n \in \mathbb{N}^*$

$$fibo(n) = \begin{cases} 1 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fibo(n-1) + fibo(n-2) & \text{sinon} \end{cases}$$

Code source 100 – Fonction fiborec : calcul du nième terme de la suite de Fibonacci

```
1 /* pre-condition : (n >= 0)*/
2 int fiborec(int n)
3 {
4         if (n == 0) {return 1;}
5         else if (n == 1) {return 1;}
6         else {return (fiborec(n - 1) * fibo(n - 2));}
7 }
```

16.0.3 Quand utiliser la récursivité

Un algorithme récursif est à privilégier s'il améliorer très sensiblement la lisibilité et la compréhension de la résolution par rapport au même algorithme itératif.

Cependant, la complexité des 2 formes (récursif et itératif) doit être étudiée afin de déterminer l'écart entre les 2. Si l'ordre de grandeur change de manière très défavorable pour l'algorithme récursif, il faut privilégier l'algorithme itératif.

De plus, l'implantation dans un langage de programmation va amener des contraintes comme la limite de capacité des nombres et les risque des débordements qui y sont liées.

16.0.4 Exercices

- Ecrire une fonction permettant le calcul de la somme des n premiers nombres entiers naturels : en utilisant un algorithme itératif, puis en utilisant un algorithme récursif et décrire l'arbre d'appels pour S(4)

[14] Dans un programme, un espace mémoire dynamique, la pile, permet l'empilement des résultats intermédiaires. Quand la pile déborde, un message de type "stack overflow" (débordement de pile) est généralement affiché et l'exécution du programme s'interrompt brutalement

- Ecrire une fonction permettant le calcul de la factorielle d'un nombre naturel en utilisant un algorithme itératif, puis en utilisant un algorithme récursif et décrire l'arbre d'appels pour F(4)

- Ecrire une fonction permettant le calcul de la puissance d'un nombre a élevé à la puissance d'un autre nombre b en utilisant un algorithme itératif, puis en utilisant un algorithme récursif et décrire l'arbre d'appels pour p(2, 3)

- Ecrire une fonction permettant le calcul du plus grand commun diviseur de 2 nombres entiers positifs en utilisant un algorithme itératif, puis en utilisant un algorithme récursif et décrire l'arbre d'appels pour pgcd(12,5)

  Indication : pgcd(a,b)=a si b =0, pgcd(a,b) = pgcd(b, a mod b) sinon

- Ecrire une fonction permettant le calcul de la racine carrée d'un nombre réel positif (algorithme de Newton) en utilisant un algorithme itératif puis en utilisant un algorithme récursif

  Indication : la suite est définie par u0 = 1, un+1 = (un + a / un) / 2

- Ecrire une fonction permettant le calcul du nième terme de la suite de Fibonacci, en utilisant un algorithme itératif puis en utilisant un algorithme récursif

  Indication : la suite est définie par u0 = 0, u1 = 1, un+1 = (un-1 + un-2) pour n >=2

Neuvième partie

Complexité

17 Complexité des algorithmes

Plusieurs solutions peuvent aboutir à la résolution d'un problème. Chaque solution va produire un algorithme différent.

Une estimation de la performance relative de chacun des algorithmes devra être calculée afin de guider le choix vers le meilleur (attention : d'autres critères sont à prendre en compte, comme la maintenabilité).

La notion de complexité d'un algorithme est une mesure de son efficacité en termes de :

- complexité *temporelle* : ordre de grandeur du temps nécessaire à son exécution et de son évolution en fonction des données d'entrée (lié au nombre d'opérations élémentaires réalisées : affectations, opérations arithmétiques, comparaisons, etc.)
- complexité *spatiale* : ordre de grandeur de l'espace occupé par les données et de son évolution en fonction des données d'entrée (lié à l'espace mémoire nécessaire à l'exécution).

Cette mesure est complexe : on s'intéressera d'avantage à un ordre de grandeur permettant de comparer plusieurs algorithmes, qu'à une valeur précise de la mesure.

Il s'agit donc de déterminer une fonction représentative de la complexité en fonction de la grandeur des données fournies à l'algorithme.

Une fois l'implémentation réalisée dans un langage de programmation, la mesure pourra être validée en mesurant effectivement le temps d'exécution et l'espace mémoire utilisé

- sur une machine spécifique (avec sa propre performance généralement mesurée en flops[15])
- et un compilateur spécifique ou bien avec des paramètres de compilations spécifiques.

On détermine la fonction de complexité, f(n), en calculant le nombre d'opérations élémentaires réalisées en fonction d'une grandeur extérieure n, n étant une quantité de données d'entrée (valeur, liste de valeurs, grandeur) ; n est le paramètre de la complexité.

17.1 Notation : grand O

Fonction de complexité, grand O

On note généralement la fonction de complexité O(g(n)) ("grand O", en anglais "Big O")), où g représente la fonction grandeur de la complexité dépendant de n, grandeur des entrées exerçant une influence sur l'exécution de l'algorithme. Cette fonction est simplifiée et tient compte de l'ordre de grandeur de la variabilité.

[15] opérations flottantes (sur des nombres réels)

17.1.1 Complexité constante : O(1)

Une complexité constante indique que quel que soit la taille des données d'entrée, l'espace et/ou le temps nécessaires seront fixes.

Code source 101 – Complexité constante

```
1 int f(int n)
2 {
3         return (n*n);
4 }
```

La mesure de l'évolution du nombre d'instructions exécutées en fonction de la grandeur de n nous donne :

- pour n valant 1, 1 instruction
- pour n valant 2, 1 instruction
- pour n valant 3, 1 instruction
- pour n valant 10, 1 instruction
- pour n valant 100, 1 instruction

Elle est de la forme f(n) = a où a est une valeur constante qui dépendra de l'architecture matérielle et logiciel de l'ordinateur d'exécution, ici pour simplifier, a vaut 1.

Exemples :

- exemple 1 (ci-dessus),
- calcul du discriminant d'une fonction du 2ème degré,
- accès direct à un élément d'un tableau grâce à un indice, etc.
- accès au 1er élément d'une liste de valeur,

Quelques soient les données d'entrée, le nombre d'instructions et l'espace alloué sera constant.

17.1.2 Linéaire : O(n)

Une complexité linéaire indique que l'espace et/ou le temps nécessaires vont augmenter de même manière que la taille des données d'entrée.

Code source 102 – Complexité linéaire

```
1 int f(int n)
2 {
3         int r = 1;
4         for(i=1;i<=n;i=i+1)
5         {
6                 r = r*i;
7         }
8         return r;
9 }
```

La mesure de l'évolution du nombre d'instructions exécutées en fonction de la grandeur de n nous donne

- pour n valant 0, 0 répétition du groupe d'instructions
- pour n valant 1, 1 répétition du groupe d'instructions
- pour n valant 3, 3 répétitions du groupe d'instructions
- pour n valant 10, 10 répétitions du groupe d'instructions
- pour n valant 100, 100 répétitions du groupe d'instructions

Elle est de la forme f(n) = n .
Exemples :

- exemples 2 et 3 (ci-dessus),
- calcul de la factorielle d'un nombre,
- parcours d'une liste de valeurs dans un tableau,
- recherche d'une valeur dans un tableau non trié ; dans cet exemple, la complexité O(n) est le pire des cas ; il se peut que la valeur cherchée soit trouvée immédiatement (au mieux : O(1)), ou en moyenne il y a 50% de chance de trouver la valeur soit dans la première moitié (O(n))

Le nombre d'opérations nécessaires sera dépendant linéairement de n.

17.1.3 Quadratique : $O(n^2)$ et polynomiale (n^c)

Une complexité quadratique indique que l'espace et/ou le temps nécessaires vont quadrupler quand n va doubler ; la polynomiale que le temps sera multiplié par 2^c quand n va doubler
Exemple :

- tri de n nombres (utilisation du tri bulle dans le pire des cas),
- parcours d'un tableau/liste de n éléments pour chaque valeur d'un tableau/liste de n éléments,
- produit de 2 matrices (n^3)

Le nombre d'opérations nécessaires sera dépendant de ^{2}

17.1.4 Autres fonctions :

- Exponentielle : $O(c^n)$ Où c est une constante (exemple de la tour de Hanoï)
- Logarithmique : $O(log_b n)$ où b est une constante (exemple : recherche dichotomique => tableau trié)
- Ou toute autre fonction permettant la modélisation de la complexité

Dixième partie

Annexes

18 Codage de l'information

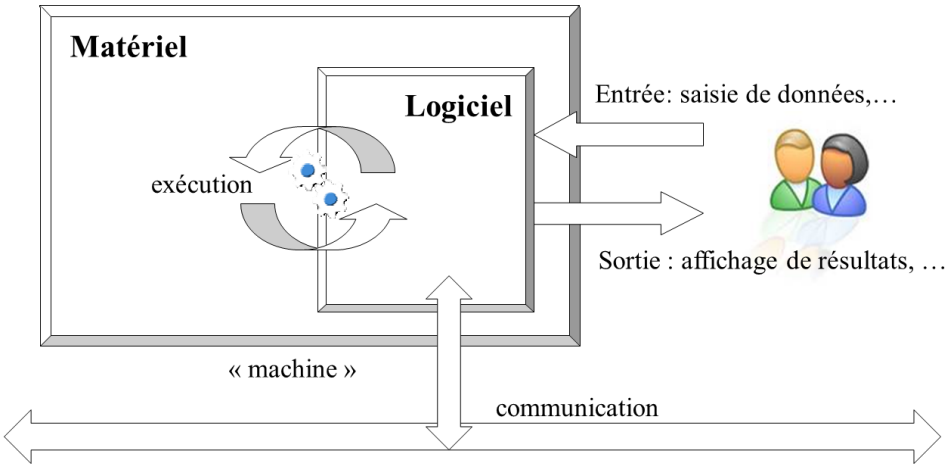
L'*informatique*, (en anglais : *IT, Information Technology*) dans le cadre de l'entreprise ou (en anglais : *Computer Science*) dans le domaine de l'enseignement, est la "*science du traitement rationnel, notamment à l'aide de machines automatiques, de l'information considérée comme le support des connaissances et des communications*".

L'information, plus précisément sa représentation, est donc au coeur du fonctionnement de l'ordinateur.

18.1 Architecture des ordinateurs

L'ordinateur est une machine (automate programmable) qui va permettre, grâce aux programmes (les logiciels qui y sont installés) et au matériel (les équipements électroniques qui le composent) d'aider l'utilisateur dans ses tâches quotidiennes en automatisant certaines de celles-ci ou en les simplifiant. (voir Figure 40 en page 133)

FIGURE 40 – Architecture simplifiée de l'ordinateur

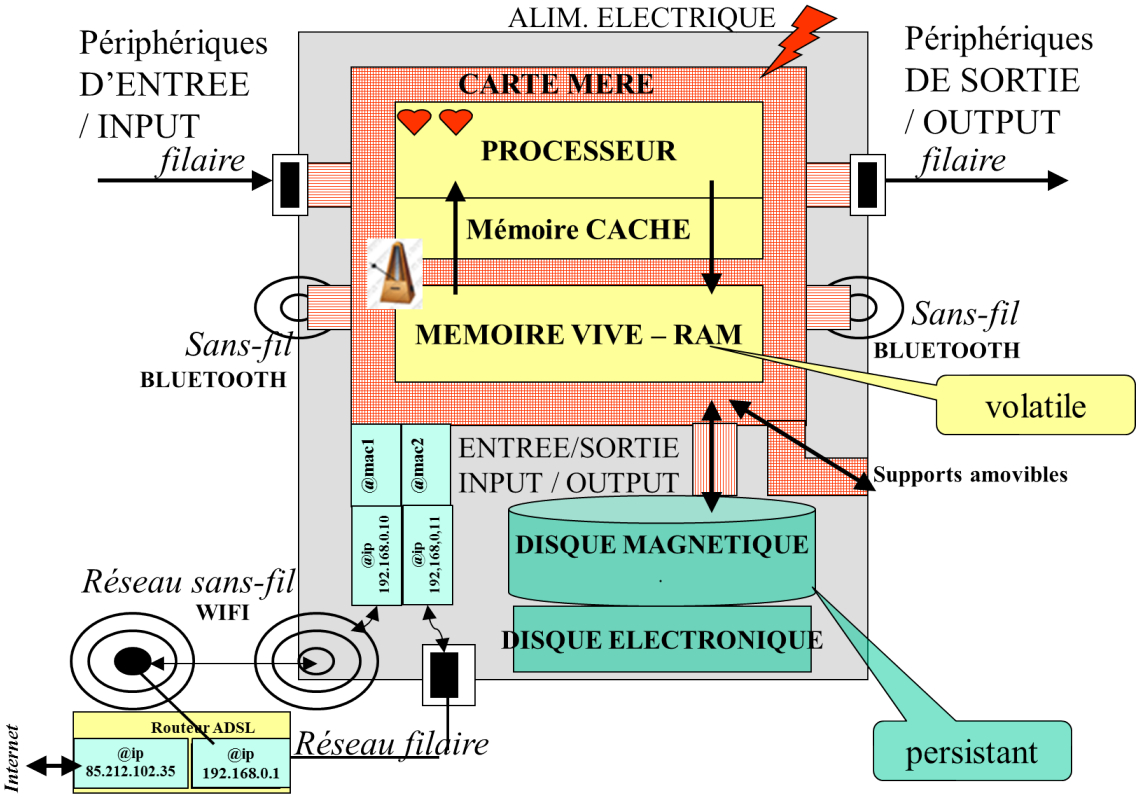


De manière plus précise, plusieurs composants essentiels sont ms en oeuvre pour assurer l'ensemble des traitements attendus par l'utilisateur. (voir Figure 41 en page 134).

L'architecture d'un ordinateur est composée de 2 ensembles formant un tout cohérent :

1. l'*architecture matérielle*, assemblage de 3 composants essentiels reliés par des *bus* :
   - l'*Unité Centrale de Traitement*, UCT, (en anglais : *CPU, Central Prossesing Unit*) : coeur de l'ordinateur, c'est le calculateur ; il est composé de :
     - l'unité arithmétique et logique (en anglais : *arithmetic-logic unit, ALU*), responsable des calculs arithmétiques, des opérations logiques, des comparaisons et autres opérations élémentaires
     - l'unité de contrôle (ou de commande) (en anglais : *control unit*) : en charge de gérer l'ordonnancement des différentes instructions à exécuter

FIGURE 41 – Architecture matérielle de l'ordinateur

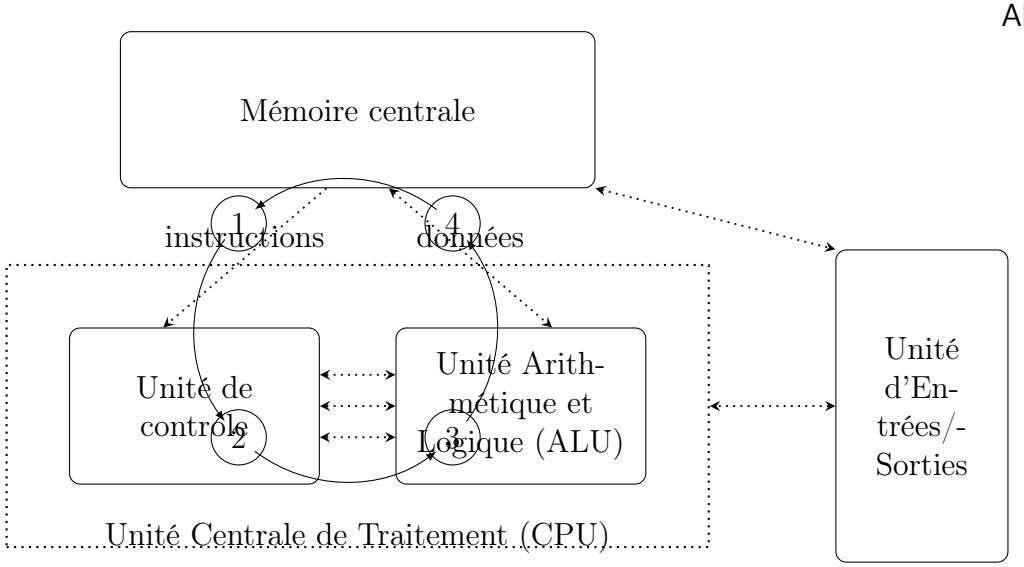


  - des registres, mémoires stockant les valeurs du calcul en cours de traitement
- la mémoire centrale (mémoire vive, appelée aussi RAM) : contient les programmes en cours d'exécution et les données en cours d'utilisation[16] ;
- les *unités d'Entrées-Sorties* (en anglais : *Input-Output units*) : assurent les échanges avec le monde extérieur (réseaux, périphériques) ;

2. l'*architecture logicielle*, assemblage de logiciels qui assurent le bon fonctionnement de l'architecture matérielle et apportent des services aux utilisateurs :
   - le *système d'exploitation* (en anglais : *Operating System*) : logiciel de base de l'ordinateur ;
   - des logiciels utilitaires : assurent des services complémentaires au système d'exploitation (pilotes de périphériques, antivirus, etc.) ;
   - des logiciels utilisateurs qui répondent aux besoin des utilisateurs.

L'exécution de chacune des instructions d'un programme (logiciel) procède de la manière suivante : : 1)récupérer l'instruction ((en anglais : *fetch*)) ; 2)la décoder ((en anglais : *decode*)) ; 3) l'exécuter ((en anglais : *execute*)) ; 4) en mémoriser le résultat. Le processeur exécute ce cycle de manière ininterrompue.

[16] les systèmes de persistance des données (mémoire de masse), comme le disque dur (en anglais : *hard disk*), assurent la conservation durable des données et programmes après que l'ordinateur ait été éteint



Tous les signaux qui circulent sur les bus et qui sont traitées par les différents composants de l'ordinateur sont électriques.

Aussi, dès la conception des 1ers ordinateurs, il a été décidé d'utiliser les symboles 0 et 1 pour représenter 2 niveaux de signaux électriques :

- ces 2 niveaux sont facilement identifiables ;
- des circuits peuvent traiter les opérations logiques (ET, OU, etc.) ;
- avec 2 valeurs élémentaires, il est aisé de représenter les nombres (binaire).

Le chiffre binaire, ou bit (en anglais : ***b****inary digi****t***) est la plus petite unité d'information manipulable par une machine numérique.

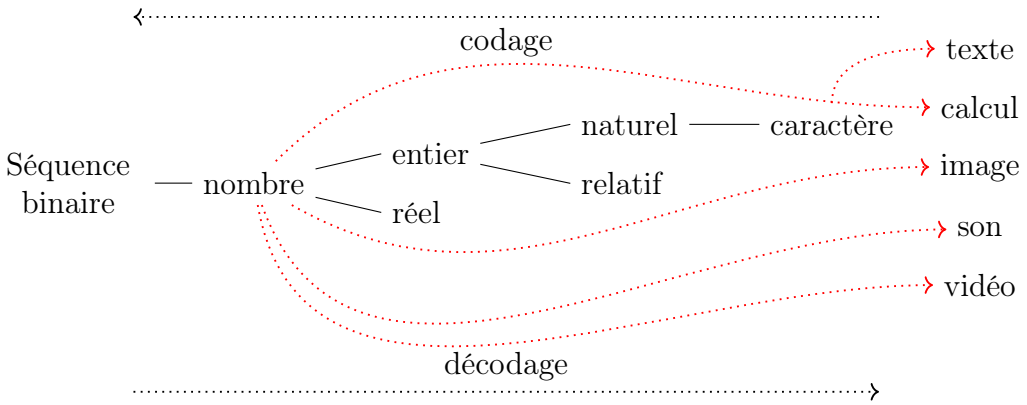
Toutes les informations traitées par l'ordinateur sont codées en une suite de 0 et de 1.

18.2 Nature et représentation de l'information

Les informations que nous utilisons possèdent de multiples formes : texte et nombres, images, sons, vidéos).

Leur traitement informatique va nécessiter

- un codage en une séquence binaire,
- puis un décodage afin de rendre le résultat intelligible pour l'être humain.



18.3 Systèmes de numération et conversion

Un *système de numération* définit la manière de représenter les nombres et comporte :

- un ensemble ordonné de symboles qui détermine la *base de numération*,
- la valeur de chaque symbole par rapport aux autres,
- des règles d'agencement de ces symboles : le rang d'un symbole dans la représentation du nombre lui donne une importance, un poids[17].

Un nombre dans une base B peut être représenté comme une succession de symboles où S_n représente le symbole S au rang n, chaque symbole ayant une valeur supérieure d'une unité[18] par rapport à son prédécesseur dans la base :

$$(\mathbf{S_n\ S_{n-1}\ ...S_2\ S_1\ S_0})_\mathbf{B}$$

plus significatif $\leftarrow$ $\rightarrow$ moins significatif

(en anglais : *most significant*) $\leftarrow$ $\rightarrow$ (en anglais : *least significant*)

Bases de numération courantes :

base		ensemble ordonné de symboles	exemple de nombre
2	binaire	0 , 1	11111100001
8	octale	0, 1, 2, 3, 4, 5, 6, 7	3741
10	décimale	0, 1, 2, 3, 4, 5, 6, 7, 8, 9	2017
16	hexadécimale	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F	7E1

[17] notation positionnelle : procédé d'écriture des nombres, dans lequel chaque position d'un chiffre ou symbole est reliée à la position voisine par un multiplicateur, qui correspond à la base du système de numération

[18] dans les systèmes de numération que nous allons utiliser

Pour compter de 0 **à** 16_{10} dans chacune des bases :

bin.	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111
oct.	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
déc.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
hex.	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

La valeur d'un nombre dans une base est obtenue en utilisant la forme polynomiale, dans laquelle chaque symbole possède un poids en fonction de son rang :

$$(S_n \times B^n + S_{n-1} \times B^{n-1} + ...S_2 \times B^2 + S_1 \times B^1 + S_0 \times B^0)_B$$

$$\text{soit : } \sum_{i=0}^{n} S_i \times B^i$$

Par exemple, la valeur décimale du nombre $(2017)_{10}$ est calculée ainsi (P = poids du symbole en fonction de son rang, S = symbole dans la base de numération) :

rang	4	3	2	1	0	
poids	10^4	10^3	10^2	10^1	10^0	
P	10000	1000	100	10	1	
$\times S$		2	0	1	7	
$=$		2000+	0+	10+	7	$= 2017$

18.3.1 Convertir d'une base B vers la base 10

On utilise la forme polynomiale :

Exemples :

$(10110)_2 \rightarrow (?)_{10}$

4	3	2	1	0
1	0	1	1	0

rang	4	3	2	1	0
poids	2^4	2^3	2^2	2^1	2^0
$(P)_{10}$	16	8	4	2	1
$\times (S)_2$	1	0	1	1	0
$=$	16	+0	+4	+2	+0

$= (22)_{10}$

$(1101111)_2 \rightarrow (?)_{10}$

6	5	4	3	2	1	0
1	1	0	1	1	1	1

rang	7	6	5	4	3	2	1	0
poids	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
$(P)_{10}$	128	64	32	16	8	4	2	1
$\times (S)_2$		1	1	0	1	1	1	1
$=$		+64	+32	+0	+8	+4	+2	+1

$= (111)_{10}$

$(FA07)_{16} \rightarrow (?)_{10}$

rang	4	3	2	1	0	
poids	16^4	16^3	16^2	16^1	16^0	
$(P)_{10}$	65536	4096	256	16	1	
$\times(S)_{16}$		F (15_{10})	A (10_{10})	0	7	
$=$		$15 * 4096$	$+10 * 256$	$+0$	$+7$	
soit		61440	$+2560$	$+0$	$+7$	$= (64007)_{10}$

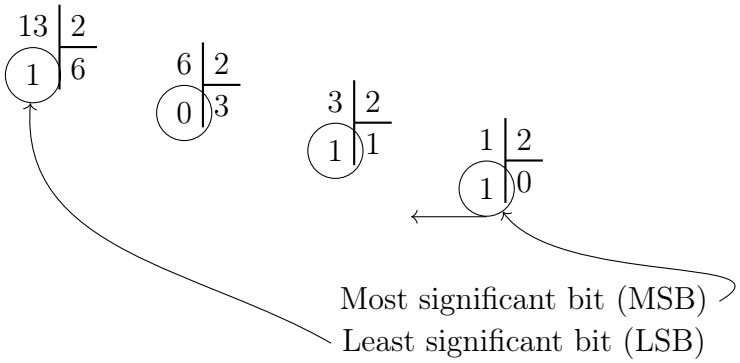
18.3.2 Convertir de la base 10 vers une base B

Deux techniques :

- divisions entières par la base pour la partie entière et multiplication par la base pour la partie fractionnaire
- utilisation d'un tableau des poids et recherche de la valeur à convertir par différences successives

18.3.2.1 Divisions entières pour la partie entière d'un nombre , jusqu'à ce que le quotient soit 0, puis récupération des restes, du dernier au premier, pour former la valeur binaire :

Conversion $(13)_{10} \rightarrow (1101)_2$



Exemples :[19]

[19] utilisation ici d'un algorithme 'optimisé' : division jusqu'à ce que le quotient soit inférieur à la base, puis récupération du dernier quotient et des restes, du dernier au premier, pour former la valeur binaire

5022	16		
4800	313	16	
222	160	19	16
160	153	16	1
62	144	3	
48	9		
14			

↓

E ←

$5022 = 139E_{16}$

5022	8			
4800	627	8		
222	560	78	8	
160	67	72	9	8
62	64	6	8	1
56	3		1	
6				

←

$5022 = 11636_8$

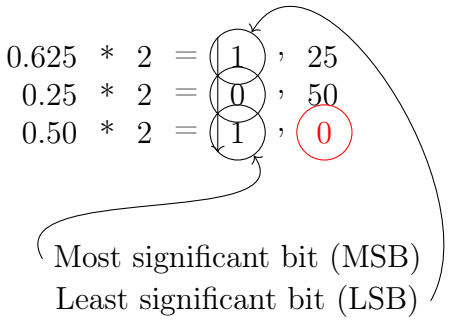
5022	2										
4000	2511	2									
1022	2000	1255	2								
1000	511	1200	627	2							
22	400	55	600	313	2						
20	111	40	27	200	156	2					
2	100	15	20	113	140	78	2				
2	11	14	7	100	16	60	39	2			
0	10	1	6	13	16	18	20	19	2		
	1		1	12	0	18	19	18	9	2	
				1		0	18	1	8	4	2
							1		1	4	2	2
										0	2	1
											0	

←

$5022 = 1001110011110_2$

18.3.2.2 Multiplications pour la partie fractionnaire d'un nombre , jusqu'à ce que la partie décimale soit 0, puis récupération des parties entières, de la dernière à la première

$$(0.625)_{10} \rightarrow (?)_2$$



$$(0.625)_{10} \rightarrow (0.101)_2$$

18.3.2.3 Utilisation d'un tableau des poids pour la partie entière ou décimale d'un nombre : recherche itérative du poids maximal inférieur ou égal à la valeur restant à trouver pour constituer la valeur dans la base

$(13.625)_{10} \rightarrow (?)_2$

rang	4	3	2	1	0	−1	−2	−3	−4
poids	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
$(P)_{10}$	16	8	4	2	1	0.5	0.25	0.125	0.0625
recherche 13.625		1							
reste 5.625			1						
reste 1.625					1				
reste 0.525						1			
reste 0.125								1	
reste 0									
binaire		1	1	0	1.	1	0	1	

$(13.625)_{10} \rightarrow (1101.101)_2$

18.3.3 Convertir de la base 2 vers la base 16 et inversement

Chaque caractère de la base hexadécimale peut être codé avec 4 bits. On utilisera cette caractéristique pour coder/décoder les valeurs binaires par groupes de 4 bits.

$(11010111)_2 \rightarrow (?)_{16}$

rang	3	2	1	0	3	2	1	0
poids	2^3	2^2	2^1	2^0	2^3	2^2	2^1	2^0
$(P)_{10}$	8	4	2	1	8	4	2	1
binaire	1	1	0	1	0	1	1	1
valeur	8	+4	+0	+1	0	+4	+2	+1
soit		13					7	
hexa		*D*					7	

$(11010111)_2 \rightarrow (D7)_{16}$

$(AF5)_{16} \rightarrow (?)_2$

rang	3	2	1	0	3	2	1	0	3	2	1	0
poids	2^3	2^2	2^1	2^0	2^3	2^2	2^1	2^0	2^3	2^2	2^1	2^0
$(P)_{10}$	8	4	2	1	8	4	2	1	8	4	2	1
hexa		A				F				5		
valeur		10				15				5		
soit :	8		+2		8	+4	+2	+1		+4		+1
binaire	1	0	1	0	1	1	1	1	0	1	0	1

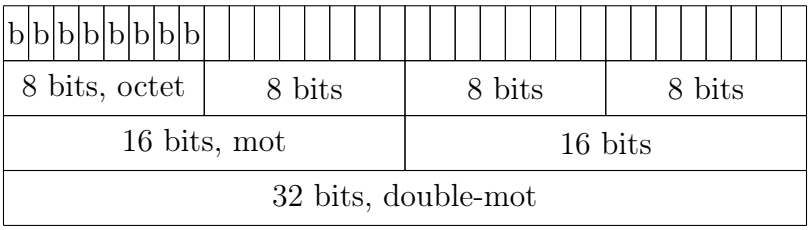
$$(AF5)_{16} \rightarrow (101011110101)_2$$

18.4 Codage de l'information

18.4.1 Les unités de stockage

Les unités de stockage de l'information binaire au sein de l'ordinateur :

- *bit* (en anglais : *binary digit*), b : information binaire élémentaire, capable de prendre 2 valeurs, 0 ou 1.
- *octet* (en anglais : *octet ou Byte*[20]), o ou B : regroupe 8 bits
- *mot* (en anglais : *word*), W : regroupe en général 2 octets
- *double-mot* (en anglais : *double word*), DW : regroupe 2 mots

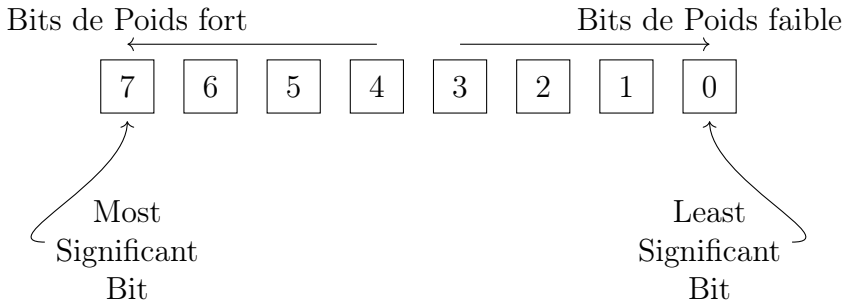


Exemples de valeurs binaires (la séparation entre les groupes de 4 bits permet une meilleure lisibilité) :

Octet	(1001 1101)
Mot	(1000 1101 1111 0001)
Double mot	(1000 1101 1111 0101 1010 1101 1011 0001)

L'octet (8 bits) est l'unité de base la plus utilisée pour définir le stockage et la manipulation des bits.

[20] un byte comporte généralement 8 bits et équivaut à un octet



Le codage de l'information est contraint par des longueurs fixes des nombres binaires.

Lorsqu'un résultat d'opération nécessite plus d'espace, il y a débordement de la capacité (en anglais : *overflow*) : le résultat n'est pas correct. Le choix de la bonne taille

18.4.2 Echelles des capacites de stockage en octets

Le Système International d'unités définit les èvpréfixes des multiples décimaux des unités :

1 kilo-octet (ko)	$= 10^3$ octets	= 1000 octets
1 méga-octet (Mo)	$= 10^6$ octets	= 1000 ko
1 giga-octet (Go)	$= 10^9$ octets	= 1000 Mo
1 téra-octet (To)	$= 10^{12}$ octets	= 1000 Go
1 péta-octet (Po)	$= 10^{15}$ octets	= 1000 To
1 exa-octet (Eo)	$= 10^{18}$ octets	= 1000 Po

Afin de prendre en compte la spécificité des unités liées au domaine numérique, il a complété sa liste et défini les *préfixes des multiples **bi**naires* , en 2008 :

1 kibi-octet (Kio)	$= 2^{10}$ octets	= 1024 octets
1 mébi-octet (Mio)	$= 2^{20}$ octets	= 1024 Kio
1 gibi-octet (Gio)	$= 2^{30}$ octets	= 1024 Mio
1 tébi-octet (Tio)	$= 2^{40}$ octets	= 1024 Gio
1 pébi-octet (Pio)	$= 2^{50}$ octets	= 1024 Tio
1 exbi-octet (Eio)	$= 2^{60}$ octets	= 1024 Pio

Exemples d'occupation mémoire par quelques valeurs simples :

- un *caractère d'un alphabet* occupe de 1 à 4 *octets* (1 octet suffit à représenter les symboles que nous utilisons en Europe, 4 octets sont nécessaires pour représenter l'ensemble des symboles des langues écrites sur la planète) ;
- un *nombre entier ou réel* occupe de 1 à 8 *octets* (selon la grandeur des valeurs qu'il est sensé contenir) ;
- un *pixel d'une image* occupe de 1 à 4 *octets* (selon la couleur : noir/blanc, 256 couleurs, etc.).
- une image occupe environ 5 Mo
- un film occupe environ 1 Go

Ces préfixes peuvent s'appliquer à toutes les unités :

1 kibibit	$= 1$ Kibit $= 2^{10}$ bit	= 1024 bit
1 kilobit	$= 1$ kbit $= 10^3$ bit	= 1000 bit

18.4.3 Nombres entiers naturels

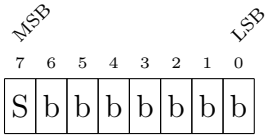
Un nombre entier naturel est codé en binaire en utilisant une des méthodes de conversion.

En fonction de sa grandeur, le codage nécessitera 1, 2 ou 4 octets (8, 16 ou 32 bits) pour sa représentation.

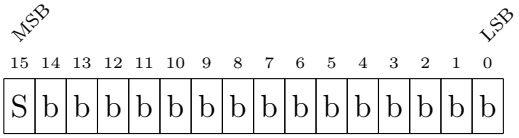
18.4.4 Nombres entiers relatifs

Plusieurs techniques sont utilisées pour coder les nombres et nécessitent une position de signe. Si la grandeur du nombre à coder nécessite plus d'un octet, c'est toujours le bit de poids fort (le plus à gauche) qui représente le signe.

Position du signe sur 1 octet



Position du signe sur 1 mot



Techniques de codage

Il existe plusieurs techniques de codage des nombres entiers relatifs. C'est celle du *complément à 2* qui est généralement mise en oeuvre.

Valeur absolue	Complément à 1	Complément à 2
7 6 5 4 3 2 1 0 0 0 0 0 0 0 1 1 vaut 3 7 6 5 4 3 2 1 0 1 0 0 0 0 0 1 1 vaut -3	7 6 5 4 3 2 1 0 0 0 0 0 0 0 1 1 vaut 3 7 6 5 4 3 2 1 0 1 1 1 1 1 1 0 0 vaut -3 ;	7 6 5 4 3 2 1 0 0 0 0 0 0 0 1 1 vaut 3 7 6 5 4 3 2 1 0 1 1 1 1 1 1 1 0 vaut -3 ;
2 valeurs du 0 : +0 (0000 0000) et -0 (1000 0000) -127 à +127	2 valeurs du 0 : +0 (0000 0000) et -0 (1111 1111) -127 à +127	Une seule valeur pour 0 : 0000 0000 -128 à +127

18.4.4.1 Pour coder un nombre entier relatif décimal en binaire

- si le nombre à coder est positif :
  1. le coder comme un entier naturel
- sinon :
  1. coder sa valeur absolue en binaire, puis calculer son complément à 2 :
     (a) effectuer le complément à 1 : inverser chaque bit (0 devient 1, 1 devient 0),
     (b) ajouter 1 au nombre binaire obtenu[21].

Exemples :

[21] On peut aussi passer au complément à 2 à partir de la valeur absolue codée en binaire en recopiant tous les 0 à partir du bit de poids faible jusqu'au premier 1 ; recopier ce 1 et inverser tous les bits suivants

$(-1)_{10} \rightarrow (?)_2$:								
$(1)_{10} \rightarrow$	0	0	0	0	0	0	0	1
$Compl.1$	1	1	1	1	1	1	1	0
$+1$								1
$= Compl.2$	1	1	1	1	1	1	1	1
$(-1)_{10} \rightarrow$	1	1	1	1	1	1	1	1

$(-1)_{10} \rightarrow (1111\ 1111)_2$

$(-23)_{10} \rightarrow (?)_2$								
$(23)_{10} \rightarrow$	0	0	0	1	0	1	1	1
$Compl.1$	1	1	1	0	1	0	0	0
$+1$								1
$= Compl.2$	1	1	1	0	1	0	0	1
$(-23)_{10} \rightarrow$	1	1	1	0	1	0	0	1

$(-23)_{10} \rightarrow (1110\ 1001)_2$

Pour décoder un nombre entier relatif binaire en décimal :

- Si le bit de poids fort est à 0 : ce nombre est positif ; on le décode comme un entier naturel
- sinon ce nombre est négatif ; on réalise l'opération de complément à 2
  - complément à 1
  - ajouter 1

  puis on utilise la forme polynomiale pour calculer la valeur de ce nombre (qui sera donc négatif)

18.4.4.2 Avantages de la codification du complément a 2 :

- une seule valeur du 0
- simplification des opérations arithmétiques : la soustraction utilise le même algorithme que l'addition

18.4.5 Codage des nombres réels

- s est le signe du nombre : + ou - ;
- m est la mantisse, nombre de l'intervalle $[1; 10[$;
- e est l'exposant, nombre entier relatif.

Exemple :

- 123456 s'écrit $+1.23456 \times 10^5$
- 0.00123 s'écrit $+1.23 \times 10^{-3}$
- -987654 s'écrit -9.87654×10^5

En faisant varier e, la position de la virgule change, 'flotte'...[22].
Représentation informatique : virgule flottante
$x = (-1)^s \times mantisse \times exposant$
$x = (-1)^s \times m \times 10^e$

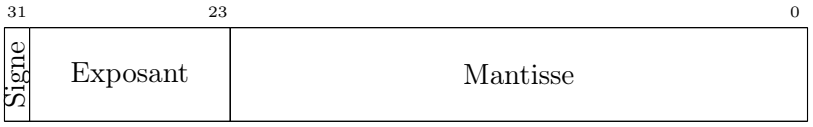
[22] d'ou le terme *virgule flottante*

18.4.5.1 Virgule flottante, format IEEE-754 La technique de codage des nombres à virgule flottante est la plus utilisée. Elle permet la représentation des nombres dans un grand intervalle de valeurs, avec cependant un inconvénient non négligeable : l'approximation du codage (et donc une perte d'information au décodage pour les nombres très grands ou très petits).

Ce codage utilise une structure bien déterminée :

- Sur 4 octets : nombres en simple précision (1 bit de signe, 8 bits pour l'exposant et 23 bits pour la mantisse)
- Sur 8 octets : nombres en double précision (1 bit de signe, 11 bits pour l'exposant et 52 bits pour la mantisse)

FIGURE 42 – [Format IEEE-754 - 32 bits]



La valeur décimale d'un nombre réel codé sur 32 bits (simple précision) est :
$x = (-1)^s \times (1.d_1d_2d_3...d_{23})_2 \times 2^{(e_1...e_8)_2-127}$
Il est codé par 32 bits de la manière suivante :

s	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}	d_{12}	d_{13}	d_{14}	d_{15}	d_{16}	d_{17}	d_{18}	d_{19}	d_{20}	d_{21}	d_{22}	d_{23}

En `double`, le nombre

$$x = (-1)^s \times (1.d_1d_2d_3 \dots d_{52})_2 \times 2^{(e_1...e_{11})_2-1023}$$

est représenté par 64 bits organisés de la manière suivante :

s	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}	e_{11}	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	$\cdots$	d_{51}	d_{52}

Exemple :

- $(A)_{IEEE}$ = 1 10000010 00110000000000000000000
- S = 1 → négatif
- e = $2^7 + 2^1 - 127 = 130 - 127 = 3$
- m = $2^{-3} + 2^{-4} = 0.125 + 0.0625 = 0.1875$
- $(A)_{10} = -1.1875 \times 2^3 = -9.5$

La somme des nombres 10.1 et 10.2 ne donne pas exactement 20.3, car la partie fractionnaire ne peut être convertie ici que de manière approchée :

```
10.1 : 0 10000010 01000011001100110011010
10.2 : 0 10000010 01000110011001100110011
20.3 : 0 10000011 01000100110011001100110
```

la somme des 2 nombres sera différente de 20.3.

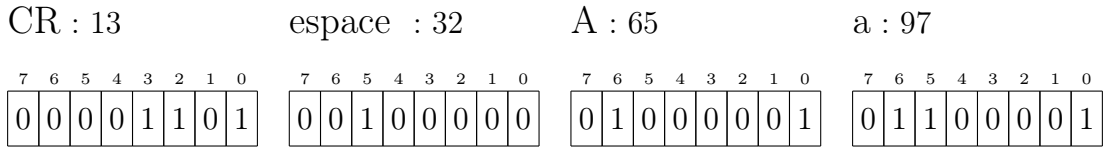
18.4.6 Représentation et codage des caractères

Un jeu de caractères (en anglais : *charset*) est un ensemble de lettres, nombres et symboles spécifiques à un pays ou une langue.

Chaque lettre est associée à une séquence binaire pour son stockage et sa manipulation : à chaque jeu de caractères correspond un tableau de correspondance entre une séquence binaire (8 bits, par exemple) et le caractère qu'elle représente.

18.4.6.1 ASCII [23]

C'est le système de codification qui était le plus utilisée jusqu'à présent. Chaque caractère utilise un octet dont le bit de poids fort est à 0 : il permet donc la représentation de 2^7 caractères codés de 0 à 127 ($2^7 - 1$).



18.4.6.2 ASCII étendu utilise le 8ème bit pour coder 128 caractères supplémentaires, mais spécifiques aux différentes langues ou régions du globe. On y trouve entre autres les caractères accentués .

18.4.6.3 UNICODE Unicode est un standard, développée par le Consortium Unicode, dont l'objectif est de définir un jeu de caractères universel : attribuer à chaque caractère de tous les systèmes d'écritures, un nom et un identifiant numérique (Universal Character Set (UCS)).

Chaque caractère unicode est un nombre compris entre 0 et $2^{31} - 1$ (soit, en hexadécimal, 0x7FFFFFFF, et en représentation décimale 2 147 483 647).

Comme chaque caractère d'Unicode ne tient pas sur le même nombre de bits, il existe plusieurs encodages pour définir les séquences de bits permettant de coder chaque caractères des UTF (Unicode Transformation Format) :

- UTF-8 : permet le codage des caractères sur un nombre de bits variables ; il est compatible avec l'US-ASCII pour les 128 1ere caractères ; chaque caractère est codé par une suite de 1 à 4 (6) octets
- UTF-16 : chaque caractère est codé par 1 ou 2 mots de 16 bits
- UTF-32 : chaque caractère est codé sur 32 bits,

[23] American Standard Code for Information Interchange

Le système de codage UTF-8 de la norme Unicode, utilise une longueur de codage variable de chaque caractère (en fonction de son numéro de code). Chaque octet définit des bits de poids fort pour distinguer l'encodage :

Caractères Unicode	Séquence binaire	Nombre de bits
0 à 127 (U-7F)	**0**bbbbbbb	7 bits
U-80 à U-7FF	**110**bbbbb **10**bbbbbb	11 bits
U-800 - U-FFFF	**1110**bbbb **10**bbbbbb **10**bbbbbb	16 bits
U-10000 - U-1FFFFF	**11110**bbb **10**bbbbbb **10**bbbbbb **10**bbbbbb	21 bits
etc.		

Chaque symbole 'b' représente une position utilisable pour coder un caractère.

18.4.7 Endianisme

L'endianisme détermine l'ordre de stockage des octets dans un système de codage nécessitant plus d'un octet et concerne :

- les nombres entiers codés sur 2 ou 4 octets,
- les nombres flottants codés en IEEE754,
- les caractères Unicode sur 2, 3 ou 4 octets

Exemple d'un nombre entier long : 67305985, qu'on peut représenter en hexadécimal par : 0x04030201

L'ordre de stockage en mémoire à l'adresse ADR va varier en fonction de l'endianisme : celui-ci dépend de l'architecture du processeur utilisé.

18.4.7.1 Big Endian (ou grand boutiste ou grand boutien) : positionne l'octet de poids fort d'abord ; il est utilisé par les architectures à base de processeur Sun, Motorola

adr	adr+1	adr+2	adr+3
04	03	02	01

18.4.7.2 Little Endian (ou petit boutiste ou petit boutien) : positionne l'octet de poids faible d'abord ; il est utilisé par par les architectures à base de processeur Intel, Alpha.

adr	adr+1	adr+2	adr+3
01	02	03	04

Pour certaines autres architectures, ce paramètre peut être positionné par logiciel.

18.5 Conclusion

Toute séquence binaire dans l'ordinateur peut représenter n'importe lequel des types de données présentés ci-dessus.

Pour décoder une séquence binaire, il est indispensable de connaitre son type (entier naturel ou relatif, réel) et la taille sur laquelle elle est codée (octet, mot, double-mot, etc.).

- un nombre entier naturel

- un nombre entier relatif
- un caractère d'un certain jeu de caractères
- un nombre réel

19 Fichier d'entêtes

Le fichier d'entête (en anglais : *header file*), à l'extension .h, rassemble un ensemble d'informations relatives à l'utilisation d'un domaine.

Il rassemble :

- Les inclusions d'entêtes
- Définition des macros
- Définition des types
- Déclaration des fonctions exportées
- Protection des multiples inclusions : le fichier d'entêtes doit contenir un ensemble de directives du pré-processeur s'assurant que son contenu ne soit pas inséré plus d'une fois [24]

Exemple de fichier 'fonctions.h'

Code source 103 – Exemple de fichier d'entêtes complet

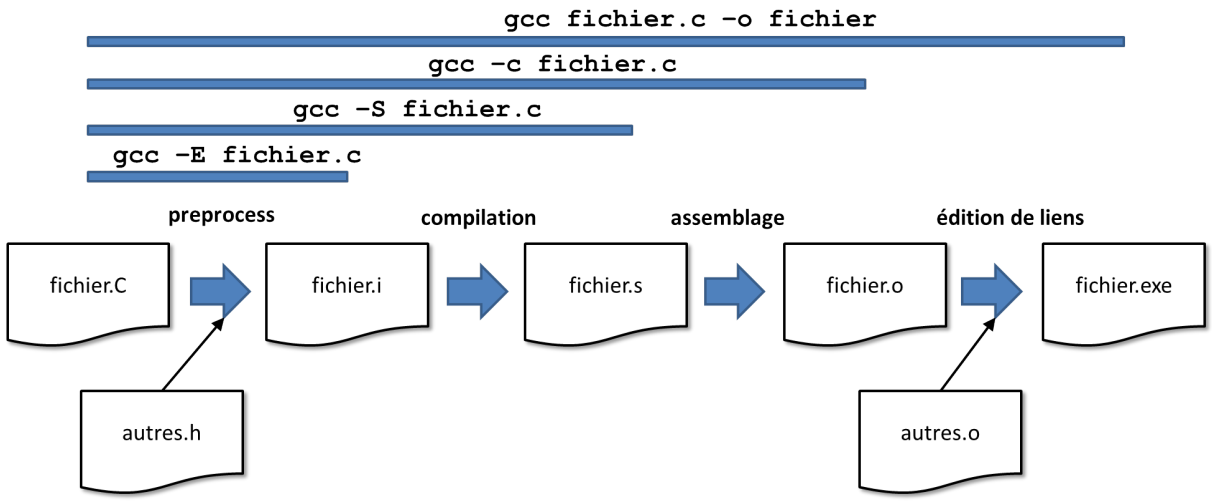
```
1  /*
2   ** Objet :
3   ** spécification des fonctions usuelles min et max
4   **
5  */
6  #ifndef FONCTIONS_H
7  #define FONCTIONS_H
8  /*
9  ** includes
10 ** ========================
11 */
12
13 /*
14 ** macros definition
15 ** ========================
16 */
17
18 /*
19 ** types definition
20 ** ========================
21 */
```

[24] ici, le choix de la condition FONCTIONS_H correspond au nom du fichier ; une autre convention rencontrée : FONCTIONS_H_INCLUDED

```
22
23 /*
24 ** exported fonctions
25 ** ====================
26 */
27
28 /*
29 ** compare 2 nombres
30 ** result = le plus grand des 2
31 */
32  int max(int a, int b);
33
34 /*
35 ** compare 2 nombres
36 ** result = le plus petit des 2
37 */
38  int min(int a, int b);
39 #endif /* FONCTIONS_H */
```

20 Chaine de compilation C

voir figure 20 page 149.



21 Nombres pseudo-aléatoires

Deux fonctions permettent la gestion des nombres pseudo-aléatoires :

- `srand` : initialise le début de la séquence aléatoire
- `rand` : retourne un nombre entre 0 et RAND_MAX

Leur déclaration se trouve dans l'entête stdlib.h

```c
void srand(unsigned int seed);
int rand();
```

ou :

- srand et rand : nom des fonctions
- seed : nombre utilisé pour l'initialisation de la fonction de génération aléatoire

```c
srand((unsigned)time(NULL));
int i = 0;
for (i=1;i<=10;i++) {
  printf("␣%d", rand());
}
```

Résultat obtenu :

```
14259 26403 30648 12347 13087 9117 11092 31364 17447 17177
```

22 Encodages des caractères

22.1 ASCII

L'ASCII est un standard américain codant 128 caractères (on parle souvent de l'US-ASCII). Il contient l'alphabet de base des langues anglo-saxonnes (lettres, caractères de ponctuation) et un certain nombre de caractères utilisés dans le contrôle des communications avec les périphériques.

ASCII CONTROL CODE CHART

b7 b6 b5 / BITS b4 b3 b2 b1	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
	CONTROL		SYMBOLS NUMBERS		UPPER CASE		LOWER CASE	
0 0 0 0	0 NUL 0	16 DLE 20	32 SP 40	48 0 60	64 @ 100	80 P 120	96 ‘ 140	112 p 160
0 0 0 1	1 SOH 1	17 DC1 21	33 ! 41	49 1 61	65 A 101	81 Q 121	97 a 141	113 q 161
0 0 1 0	2 STX 2	18 DC2 22	34 " 42	50 2 62	66 B 102	82 R 122	98 b 142	114 r 162
0 0 1 1	3 ETX 3	19 DC3 23	35 # 43	51 3 63	67 C 103	83 S 123	99 c 143	115 s 163
0 1 0 0	4 EOT 4	20 DC4 24	36 $ 44	52 4 64	68 D 104	84 T 124	100 d 144	116 t 164
0 1 0 1	5 ENQ 5	21 NAK 25	37 % 45	53 5 65	69 E 105	85 U 125	101 e 145	117 u 165
0 1 1 0	6 ACK 6	22 SYN 26	38 & 46	54 6 66	70 F 106	86 V 126	102 f 146	118 v 166
0 1 1 1	7 BEL 7	23 ETB 27	39 ' 47	55 7 67	71 G 107	87 W 127	103 g 147	119 w 167
1 0 0 0	8 BS 10	24 CAN 30	40 (50	56 8 70	72 H 110	88 X 130	104 h 150	120 x 170
1 0 0 1	9 HT 11	25 EM 31	41) 51	57 9 71	73 I 111	89 Y 131	105 i 151	121 y 171
1 0 1 0	10 LF 12	26 SUB 32	42 * 52	58 : 72	74 J 112	90 Z 132	106 j 152	122 z 172
1 0 1 1	11 VT 13	27 ESC 33	43 + 53	59 ; 73	75 K 113	91 [133	107 k 153	123 { 173
1 1 0 0	12 FF 14	28 FS 34	44 , 54	60 < 74	76 L 114	92 \ 134	108 l 154	124 | 174
1 1 0 1	13 CR 15	29 GS 35	45 − 55	61 = 75	77 M 115	93] 135	109 m 155	125 } 175
1 1 1 0	14 SO 16	30 RS 36	46 . 56	62 > 76	78 N 116	94 ^ 136	110 n 156	126 ~ 176
1 1 1 1	15 SI 17	31 US 37	47 / 57	63 ? 77	79 O 117	95 ¨ 137	111 o 157	127 DEL 177

LEGEND : dec CHAR hex oct

Victor Eijkhout
Dept. of Comp. Sci.
University of Tennessee
Knoxville TN 37996, USA

22.2 ISO-8859-1 ou latin1

Cet encodage complète l'ASCII de 128 caractères encodant la majorité[25] des caractères des langues de l'Europe de l'ouest.

[25] Certains caractères comme le oe français ne sont pas représentés.

22.3 ISO-8859-15 ou latin9

Cet encodage modifie légèrement le latin1 et y ajoute le symbole euro et le oe français.

22.4 CP1252

C'est un encodage Windows, une extension du latin1, mais dont certains caractères sont différents.

	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E_	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
	00E0	00E1	00E2	00E3	00E4	00E5	00E6	00E7	00E8	00E9	00EA	00EB	00EC	00ED	00EE	00EF
	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

FIGURE 43 – Extrait du code page 1252

22.5 CP850

C'est un encodage Windows utilisé en mode commande et qui présente certaines différences par rapport au CP1252 :

	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E_	Ó	ß	Ô	Ò	õ	Õ	µ	þ	Þ	Ú	Û	Ù	ý	Ý	¯	´
	00D3	00DF	00D4	00D2	00F5	00D5	00B5	00FE	00DE	00DA	00DB	00D9	00FD	00DD	00AF	00B4
	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
	SHY	±		¾	¶	§	÷		°	¨	·	1	3	2	■	NBSP

FIGURE 44 – Extrait du code page 850

22.6 Une normalisation

22.6.1 Norme ISO/CEI 10646

Cette norme[26] a pour but de définir un système de codage universel pour tous les systèmes d'écriture de la planète. Cette norme est le fondement du standard Unicode.

22.6.2 Standard Unicode

Unicode est un standard[27] informatique qui permet des échanges de textes dans différentes langues, à un niveau mondial.

Unicode comporte essentiellement

- un répertoire de caractères qui associé au caractère un nom en clair ;

[26] Définition officielle ISO : "Document établi par un consensus et approuvé par un organisme reconnu, qui fournit, pour des usages communs et repérés, des règles, des lignes directrices ou des caractéristiques, pour des activités ou leurs résultats, garantissant un niveau d'ordre optimal dans un contexte donné.". En anglais : standard !

[27] Ensemble de recommandations développées et préconisées par un groupe représentatif d'utilisateurs.

- un index numérique associé à chaque caractère ;
- 3 systèmes d'encodage des caractères :
  - UTF-8 sur 1 à 4 octets ;
  - UTF-16 sur 2 ou 4 octets ;
  - UTF-32 sur 4 octets.

23 Risques liés à l'usage des macros

Des risques sont inhérents à l'usage des macros avec substitution ; aussi est-il indispensable de réfléchir à leur remplacement (par des constantes ou des fonctions) ou de s'assurer des effets de bord pouvant être produits par leurs définitions.

Ce type de macros permettent l'introduction de variables dans la substitution : ici, par exemple, une valeur `carre(2)` sera remplacée par `(2*2)` : cette possibilité parait intéressante mais exige d'être très vigilant sur les conséquences possibles.

1. substitution initiale :

```
1 #define carre(x) x*x
```

   Mais attention :

```
1 2/carre(10)
```

   sera remplacée par :

```
1 2/10*10
```

   soit : 2 alors qu'on attend 0.02 !

2. Une amélioration immédiate permet de protéger contre cette anomalie :

```
1 #define carre(x) (x*x)
```

   Mais attention encore :

```
1 carre(1+1)
```

   sera remplacée par :

```
1 1+1*1+1
```

   soit : 3 alors qu'on attend 4 !

3. une nouvelle amélioration semble nous protéger :

```
1 #define carre(x) ((x)*(x))
```

   Mais attention enfin :

```
1 carre(++x)
```

qui sera remplacée par :

```
1 (++x)*(++x)
```

soit, si x vaut 2, vaudra 16 alors qu'on pourrait attendre 9 !

Le compilateur peut indiquer le risque d'anomalie : exemple gcc "warning : operation on 'x' may be undefined [-Wsequence-point]"

Sur ce dernier exemple, se souvenir également que l'utilisation des opérateurs d'incrémentation (++ et --) sont à proscrire dans des expressions (calculs, comparaisons, appels de fonctions, etc.)

Conseil

Les macros avec substitution et arguments ne devraient être utilisées uniquement que lorsqu'aucune fonction ne peut être écrite pour les remplacer.

24 Risques liés à l'usage des entiers signés

Conversion `signed` vers `unsigned` Il faut être très vigilant lors de combinaison de types signés avec des types non signés.

Un exemple connu va vous le démontrer :

```
1  unsigned int a = 6;
2  int b = -20;
3
4  if ((a+b) > 0) cout << "positif : " << (a+b);
5  else cout << "negatif : " << (a+b);
6  if ((a+b) < 0) cout << "negatif : " << (a+b);
7  else cout << "positif : " << (a+b);
8  if ((a+b) > -1) cout "positif : " << (a+b);
9  else cout << "negatif : " << (a+b);
10 if ((a+b) < -1) cout << "negatif : " << (a+b);
11 else cout << "positif : " << (a+b);
```

L'exécution produit :

```
positif : -14
positif : -14
negatif : -14
negatif : -14
```

Ici la conversion de type est réalisée lors de la comparaison. Dans les 2 premiers cas, la valeur signée est convertie en `unsigned` ("promotion de type"...), est devient donc positive pour le test (cf. conversion binaire du complément à 2).

Pour plus d'infos. cf. https://www.securecoding.cert.org

La promotion de type est un outil très puissant, elle doit donc être utilisée avec prudence. Le compilateur fait de lui-même des conversions lors de l'évaluation des expressions. Pour cela, il applique des règles de conversion implicite. Ces règles ont pour but la perte du minimum d'information dans l'évaluation de l'expression.

Les rangs des types entiers, du plus grand au plus petit :

- long long int, unsigned long long int
- long int, unsigned long int
- int, unsigned int
- short int, unsigned short int
- signed char, char, unsigned char
- _Bool

Autre exemple :

```
1 int a = -16;
2
3 cout << a << "␣"
4    << static_cast<unsigned int>(a);
```

affiche :

```
1 -16
2 4294967280
```

La valeur de a est transformée en nombre positif sans pour autant modifier sa valeur interne : l'affichage produit ainsi 2 valeurs différentes.

25 Fonctions des bibliothèques standard

Les bibliothèques à utiliser sont indiquées entre parenthèses dans les sous-titres suivants.

25.1 Fonctions mathématiques : `math.h`

Le langage C met à disposition du programmeur des fonctions mathématiques standard parmi lesquelles :

Code source 104 – Prototypes des fonctions de la bibliothèdoubleque cmath

```
1 double sin (double); // sinus
2 double cos (double); // cosinus
3 double tan (double); // tangente
4 double atan (double); // arc tangente
5 double sqrt (double); // racine carrée
6 double pow (double, double); // puissance
7 double exp (double); // exponentation
8 double log (double); // logarithme naturel
9 double log10 (double); // logarithme décimal
```

25.2 Fonctions de tests de valeurs numériques (`math.h`)

Le langage C met à disposition du programmeur des fonctions permettant d'évaluer des expressions afin de prévoir un éventuel débordement :

Code source 105 – Prototypes des fonctions de test d'expressions

```
1 bool isinf(expression); // test expression infinie (trop grande) 1/0
2 bool isnan(expression); // test expression n'est pas un nombre cohérent (vale
3 bool isfinite(expression); // test expression ni inf ni nan
```

où *expression* est le calcul à évaluer (nombre réel).

26 Instruction `assert` et mode déboguage

L'instruction `assert` introduit une assertion (expression dont on attend a ce qu'elle soit vraie) et permet de vérifier ces assertions à certains points stratégiques du code.

Elle peut être désactivée de 2 manières :

- en ajoutant la directive suivante dans les codes sources où elle est utilisée (et avant l'inclusion de la bibliothèque assert) :

Code source 106 – Macros pour desactiver l'instruction assert

```
1 #define NDEBUG
```

- en ajoutant un paramètre à la commande gcc lors de la phase de compilation

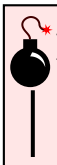
Code source 107 – Desactiver l'instruction assert lors de la compilation

```
1 gcc -Wextra -Wall -NDEBUG -c source.cpp
```

La définition d'une macro spécifique va permettre un contrôle indépendant du mode debug.

Code source 108 – myassert.h

```
1 #ifndef MY_ASSERT
2 #define my_assert(expr) do { \
3   if (!(expr)) { \
4     fprintf(stderr, "Assertion %s in %s(%d) failed\n", #expr, __FILE__, __LINE_
5     abort(); \
6   } \
7 } while(0)
8 #endif
```



Danger

Les assertions représentent des forme de contrat de bonne exécution d'une fonction si la pré-condition est remplie.

Le programmeur doit absolument prévoir d'effectuer l'appel avec les bons arguments et donc vérifier ces valeurs auparavant (contrôle des saisies, etc.)

27 Langages de programmation

Il existe de nombreux langages de programmation. Ils ont été conçus afin de répondre à plusieurs enjeux :

- l'évolution des technologies liées à l'informatique (puissance des ordinateurs, capacité de la mémoire vive) : au départ liés à l'aspect matériel des ordinateurs (binaire), l'expression s'est voulue ensuite plus proche du langage naturel ;
- l'expression de besoins particuliers : gestion, calcul scientifique, enseignement, systèmes d'exploitation, graphisme, intelligence artificielle, etc. ;
- une évolution dans la manière d'appréhender les problèmes et de concevoir les programmes (*paradigmes de programmation*).

On peut classer les langages de programmation selon plusieurs critères pour constituer des familles de langages :

- générations de langages : proximité plus ou moins grande du langage machine ;
- paradigmes de programmation : de quelle manière les problèmes seront-ils appréhendés et les programmes concus ?
- mode d'exécution : comment les instructions d'un programme sont-elles exécutées ?

27.1 Générations de langages

On distingue 4 générations de langages :

L1G, Langages de 1ère génération : langage machine dans lequel chaque instruction est codée sous forme binaire, une succession de 0 et de 1 ; c'est le langage de base compris par les processeurs[28]

Exemple de langage binaire (simulateur Mars) :

```
1 0000 0010 0001 0001 1000 0000 0010 0000
```

L2G, Langages de 2nde génération : à chaque instruction de base est associé un code mnémonique ; ce sont les langages assembleurs (un par famille de processeurs...)

Exemple de langage assembleur (simulateur Mars) :

```
1 add $s0 ,$s0 ,$s1 # additionner $s0 et $s1 , ranger dans $s0
```

[28] Chaque famille de processeurs dispose d'un jeu d'instructions : ensemble des instructions qui peuvent être codées pour cette famille.

Par curiosité...
Tapez "Intel pentium instruction set" sur un moteur de recherche : vous aurez accès à la documentation des jeux d'instruction des différents processeurs Intel.

L3G, Langage de 3ème génération : le langage devient évolué, la syntaxe se rapproche du langage naturel ; on parle de *langage de haut niveau* : Cobol, C, Pascal, Basic, C++, Java, C#, Visual Basic, Ada, etc. ; une *révolution* vient de s'opérer : le code source devient indépendant d'une quelconque architecture matérielle puisqu'un compilateur (un par famille de processeurs) va traduire les instructions dans le code machine cible ;

Exemple source Cobol :

```
01 a PIC S9(6)V99 VALUE 1000.
01 b PIC S9(6)V99 value 3.
01 c PIC S9(6)V99.
...
COMPUTE c ROUNDED = (a / b)
  ON SIZE ERROR PERFORM erreur.
```

Exemples source C, C++, Java :

```
float a, b, c;
a = (b + c);
```

L4G, Langages de 4ème génération : des macros instructions permettent l'écriture rapide de programmes : RAD (en anglais : *Rapid Application Development*) ;

Exemple du Wlangage, le langage de Windev :

```
MatRéelAdditionne(NomMatrice, 5)
```

27.2 Paradigmes de programmation

Les paradigmes de programmation correspondent à différentes manières d'appréhender la résolution des problèmes et de mettre en oeuvre l'organisation des données et des instructions pour écrire les codes sources d'un programme.

En programmation impérative , on indique à l'ordinateur la séquence des instructions à exécuter : l'exécution va faire évoluer les données du problème vers sa résolution (l'état des variables-mémoire représente l'évolution vers le résultat attendu, ce qui peut provoquer des effets de bord, modifications qui peuvent avoir un effet sur une autre partie de l'exécution) ; on y décrit le "comment faire".

- La *programmation structurée* : on conçoit un programme comme une séquence d'instructions dont l'exécution est organisée à l'aide de structures de contrôles - choix, répétitions - (C, Cobol, Pascal, Basic, Fortran)
- La *programmation procédurale* : on conçoit un programme comme un ensemble de procédures et fonctions (C, Cobol, Pascal, Basic, Fortran)

- La *programmation orientée objets* : on conçoit un programme comme un ensemble d'objets échangeant des messages (Eiffel, Smalltalk, Java, C++, C#, Pascal Objet)

Calcul de la factorielle d'un nombre (C) :

```c
int factorielle ( int n ) {
  int res = 1;
  int i = 0;
  for ( i = 2; i <= n; i++ ) {
    res = res * i;
  }
  return res;
}
```

En programmation fonctionnelle , on indique à l'ordinateur la séquence des instructions à exécuter comme un ensemble de fonctions qui manipulent des ensembles de données constitués par d'autres fonctions (fonctions d'ordre supérieur - utilisent d'autres fonctions comme arguments -, les appels sont récursifs, pas d'effet de bord) (Haskell, Lisp, Scheme, Caml, Scala, etc.). L'opération d'affectation n'est pas utilisée, les variables étant immuables (non modifiables) : pas de modification de l'état mémoire = pas d'effet de bord.

Calcul de la factorielle d'un nombre (Scheme) :

```
(define (fac n)
 (if (= n 0)
   1
   (* n (fac (- n 1)))))
```

En programmation déclarative , un ensemble de déclaration sont énumérées (description du résultat attendu ou de règles à appliquer aux données sans préciser la séquence d'exécution)

- langage d'interrogation des bases de données relationnelles (SQL)
- langage d'interrogation/transformation de documents XML : XSLT
- en programmation logique, un programme est une suite de déclaration de faits et de règles ; un "moteur d'inférence" déduit des conclusions (= nouveaux faits) à partir de ces règles (Prolog)

27.3 Modes d'exécution des programmes

Le mode d'exécution définit la manière dont les instructions d'un programme sont exécutées :

- langage *interprété* : les instructions du code source sont traduites en langage machine au fur et à mesure de l'exécution ; l'exécution est (relativement) lente ; exemples : les langages de script (Javascript, VBscript, bash, etc.), Python, PHP, etc.
- langage *compilé* : les instructions du code source doivent être d'abord traduites en langage machine avant d'être exécutées ; l'exécution est la plus rapide ; exemples : la plupart des langages de programmation (C, C++, Ada, etc.).

- langage *à code intermédiaire* : les instructions du code source sont traduite dans un langage intermédiaire ; au moment de l'exécution, un programme particulier, appelé "machine virtuelle" (en anglais : *VM, Virtual Machine*) se charge de convertir ce code intermédiaire en langage machine ; l'exécution est relativement rapide (Java, C#, VB.net, etc.)