1 Eléments de base

1.1 Introduction à la programmation impérative

L'objet de ce cours est d'apprendre à programmer un ordinateur, *i.e.* d'apprendre à décrire les opérations que la machine doit effectuer pour résoudre automatiquement un problème. Il s'agit donc d'apprendre à écrire un programme et pour cela, il nous faut introduire un langage de programmation.

Dès lors que les problèmes à résoudre deviennent complexes (au-delà des opérations arithmétiques élémentaires), programmer directement en langage machine devient impossible (lourdeur des programmes, niveau de détail empêchant de mettre en œuvre des raisonnements complexes, nécessité de tenir compte des caractéristiques propres de la machine...). On utilise alors un langage de programmation évolué. Langages formels et non ambigus, les langages de programmation évolués obéissent à une syntaxe stricte comme le langage machine. Ils offrent néanmoins une plus grande richesse d'expression qui permet au programmeur de s'affranchir d'un certain niveau de détail. Langages compilables, ils permettent aussi d'écrire des programmes indépendants de la machine sur laquelle ils seront exécutés.

Dans ce cours, nous utiliserons le langage C. Il appartient à la famille des langages impératifs (Fortran, Cobol, Pascal, Basic, etc.), par opposition aux langages fonctionnels ou objet. Le langage C a été conçu à l'origine aux Laboratoires Bell (en même temps que le système Unix) pour faire de la programmation système. Il est très efficace et très utilisé aujourd'hui dans l'industrie. Un certain nombre de langages actuels, comme C++, Perl, ou Java, dérivent également de C et présentent avec lui des similitudes syntaxiques.

On peut écrire des programmes dans de nombreux langages. L'objectif ici est pas d'apprendre à programmer en C. Il s'agit de maîtriser les bases de la programmation impérative et de les mettre en œuvre dans une programmation en langage C. Nous emploierons donc souvent un langage algorithmique indépendant de C qui nous permettra :

- de souligner les étapes importantes d'un algorithme, indépendamment des détails de son implémentation ;
- de maîtriser les concepts de la programmation impérative indépendamment des spécificités du C ou de tout autre langage particulier ;
- de souligner les particularités de C.

Rappelons en effet les trois premières phases de la conception d'un logiciel :

1. L'**analyse,** qui permet de spécifier le problème à résoudre et/ou d'isoler les fonctionnalités du logiciel à mettre en oeuvre. Dans cette phase on détermine « quelles sont les données et leurs propriétés, quels sont les résultats attendus et quelles sont les relations exactes entre les données et les résultats » (cf. Bibliographie [2]).
2. La **conception** de l'algorithme ; il s'agit généralement de déterminer la méthode de résolution d'un problème, d'en identifier les principales étapes et de la décrire dans ses grandes lignes. C'est en réalité dans cette phase qu'on spécifie la manière dont on va mettre

en œuvre les fonctionnalités du logiciel que l'on a isolé dans la phase d'analyse. A ce stade, certains détails peuvent encore être laissés de côté, mais il faudra veiller à ne laisser aucun point important dans l'ombre.

3. L'**implémentation** (ou **implantation**) de l'algorithme dans un langage de programmation (C dans notre cas). Il s'agit de traduire la conception de l'algorithme en un programme dans un langage de programmation particulier. A ce stade, il faut respecter la **syntaxe** du langage de programmation choisi et préciser tout ce qui était resté dans l'ombre. la syntaxe est l'ensemble des règles d'écriture d'un langage qui détermine l'« allure » et la structure des programmes. Cette syntaxe doit être rigoureusement suivie. Une seule virgule mal placée et le programme ne peut pas être exécuté (le compilateur ne génère pas de programme exécutable), ou pire, il « tourne » mais ne fait pas ce qui était attendu (on a des erreurs pendant l'exécution ou des résultats faux).

Nous mettrons ici l'accent sur la seconde et la troisième de ces étapes.

1.2 Notations

- Les exemples décrits en langage algorithmique figurent en italiques.
- Les exemples de programmes ou les instructions C figurent dans ce document en police `courier`.
- Sont notés en gras les éléments du langage nouvellement introduits ou les éléments qui font ressortir la structure des programmes.
- Les crochets (<…>) sont utilisés dans la description de la syntaxe du langage pour désigner de manière générique des éléments à remplacer. On peut ainsi décrire une déclaration de fonction entière par la formule : `int <identificateur de fonction>() ;` qui indique que l'on peut mettre à la place de ce qui est entre crochets n'importe quel identificateur de fonction (par ex. : `int maFonction() ;`).
- Le texte souligné représente les messages d'entrées (i.e. tapés par l'utilisateur au clavier) et les messages de sorties (affichés à l'écran).

1.3 Actions / conditions

Considérons deux exemples :

- Une calculatrice de bureau dont on suppose dans un premier temps qu'elle ne fait que des additions. L'addition consiste à récupérer la première et la seconde valeur donnée par l'utilisateur (1ère et 2nde opérandes), à en faire l'addition et enfin, à afficher le résultat.
- Une billetterie automatique pour l'achat de billets de train. Le dialogue de l'utilisateur avec la billetterie permet de préciser de quel trajet il s'agit, de fixer la date et l'horaire, de déterminer le tarif auquel l'utilisateur a droit, et de calculer le prix du ou des billets demandés.

Nous observons sur ces exemples que des actions complexes (ici, une addition ou un dialogue de billetterie automatique) se décomposent en une séquence d'actions plus simples. Nous verrons plus loin en quoi consistent dans un programme ces sous-actions.

Exemple : addition

Algorithme
2 variables entières en entrée : n1, n2
1 variable entière en sortie : r
début
lecture de la première variable: n1
lecture de la seconde variable: n2
calcul du résultat : r = n1 + n2
affichage du résultat : r
fin algorithme

Exemple : billetterie

Algorithme
début
affichage du message d'accueil (« vous êtes sur une billetterie automatique »)
choix du trajet
calcul du prix de base en fonction du trajet :p
choix du nombre de billets : nb
*calcul du montant total : m = p * nb*
affichage du résultat : m
fin algorithme

Si maintenant, on considère l'opération de division, il faut vérifier avant de procéder au calcul que le diviseur n'est pas nul. Cette fois l'action correspondant au calcul est conditionnée par un test sur la valeur du diviseur.

Algorithme : division

Algorithme
2 variables entières en entrée : n1, n2
1 variable entière en sortie : r
début
lecture de la première variablel: n1
lecture de la seconde variable: n2
si (n2 ≠ 0)
alors calcul du résultat : r = n1 / n2
affichage du résultat : r
finsi
fin algorithme

Ces algorithmes représentent de manière abstraite des programmes. Ils sont constitués d'une suite d'instructions à effectuer, comme nous l'avons vu en MAMIAS. Une instruction peut être simple ou conditionnelle. La lecture d'une variable est ici une instruction simple. Le deuxième cas est introduit par l'instruction « si » ; dans ce cas, l'action (ou les actions) commandées par le « si » ne seront exécutées que si la condition (ici n2 ≠ 0) est vérifiée.

1.4 Valeurs, types et variables

1.4.1 Valeurs et types

Un programme manipule des valeurs. Ces valeurs étant codées en machine, leurs représentations machine seront des suites de 0 et de 1.

Les valeurs sont typées. Un type définit l'ensemble abstrait que des valeurs peuvent prendre. Le type d'une valeur détermine également les opérations que l'on peut effectuer avec cette valeur. On distingue par exemple le type numérique entier et le type caractère imprimable. La « signification » d'une valeur représentée par une suite de 0 et de 1 dépendra donc du type d'information dont il s'agit. Par exemple, le code 01001111 représente aussi bien l'entier non signé 79, que l'entier signé +79 (codage sur 8 bits) ou le caractère 'O' (si l'on sait qu'un caractère est codé par un entier : son code Ascii, ici 4F).

Il existe plusieurs types classiques dans les langages de programmation : des types numériques (entiers, réels, etc.), le type booléen (il n'y a que deux valeurs : le VRAI et le FAUX) et le type caractère. Nous introduiront formellement plus loin ceux utilisés dans le langage C. Pour le moment, nous ne manipulons que des valeurs de nombres entiers (ou valeurs *entières*).

1.4.2 Variables

Les valeurs sont stockées dans des zones (ou « cases ») de la mémoire. Chaque zone a une adresse mais, dans un langage de programmation évolué, on n'accède pas aux zones de mémoire directement par leurs adresses. On préfère manipuler des **variables**.

Une variable est une zone de la mémoire centrale qui possède un nom, appelé **identificateur**, permettant de la désigner. Cet identificateur est donc associé à une **adresse** mémoire (ou référence), mais cette adresse reste généralement inconnue du programmeur.

Pour pouvoir interpréter le contenu situé en mémoire, on associe également à une variable un **type**.

L'**évaluation** de l'identificateur de la variable renvoie la **valeur** contenue à l'adresse mémoire correspondante. Cette valeur est interprétée en fonction du type de la variable.

Pour conclure, une variable se définit par l'association d'une adresse (sa référence), d'un identificateur (son nom), et d'un type. Le nom permet d'accéder à l'adresse, l'adresse au contenu, et le type permet d'interpréter le contenu comme une valeur.

a. Identificateur

Un identificateur est un nom. Nous verrons plus loin qu'on utilise les identificateurs comme noms pour des variables, des types, des constantes, des fonctions, etc. De manière générale, un identificateur est un nom qu'on attribue à une entité qu'on définit. Mais il existe déjà dans un langage de programmation un certain nombre de mots ou caractères qui ont déjà une signification. Ces mots s'appellent des **mots réservés**. On ne peut pas choisir comme identificateur de variable ou de fonction un mot réservé du langage.

Identificateurs et mots réservés en C

Identificateurs	***mots réservés***
Noms composés de lettres majuscules ou minuscules, de chiffres, et du caractère souligné '_'.	`main, if, else, while, auto, register, extern, int, …`

Exemples

`b`, `billet`, `billet_retour`, `billetRetour`, `a`, `b`, `c`, sont des identificateurs valides. Par contre, `question?` , `auto` ou `#fg56` ne sont pas valides. Dans l'algorithme ci-dessus, trois variables sont utilisées. Leurs identificateurs sont respectivement `n1`, `n2`, et `r.`

Le choix d'une convention pour l'écriture des identificateurs améliore la lisibilité et la compréhension des programmes. Il existe en C des conventions de nommage qui sont recommandées : ainsi, on fera commencer les identificateurs de variables ou de fonctions par une minuscule.

Il vaut mieux choisir des identificateurs explicites même s'ils sont un peu longs. Dans ce cours, nous éviterons les caractères '_' dans les identificateurs et nous emploierons des majuscules pour séparer les mots quand un identificateur en comprend plusieurs.

Parmi les identificateurs ci-dessus, on préfère `billet`, et `n1` aux identificateurs `b` et `i`, qui sont moins explicites. (On réservera plutôt l'usage de `i` et `j` à des variables entières dans des boucles). On préfère par ailleurs `billetRetour` à `billet_retour`. On évitera également les identificateurs terminés par un zéro, comme num0, ou ceux notés O1 et O2, à cause des confusions possibles entre le chiffre zéro et la lettre O. Une faute de frappe est en effet ensuite difficile à repérer.

De manière générale, un identificateur doit toujours être le plus parlant possible (tout en restant court) ; il doit être lié au vocabulaire du problème analysé (i .e. celui utilisé dans la première phase d'analyse ou dans les documents de conception). C'est tout un art que de trouver de bons identificateurs, et c'est à cette capacité qu'on reconnnaîtra un bon programmeur.

b. Adresses

Un langage de programmation évolué ne manipule pas directement les adresses où sont stockées les données en mémoire mais des variables. Il arrive cependant en C qu'on doive faire référence à l'adresse d'une variable. Dans ce cas, on ne peut manipuler directement le code de l'adresse, puisqu'il varie d'une exécution à l'autre, mais on peut utiliser l'opérateur d'adresse, noté `&,` qui, placé devant un identificateur de variable, désigne l'adresse de cette variable.

Exemple

Si `v` est une variable d'un programme C, l'évaluation de `v` retourne (ou renvoie) sa valeur alors que celle de `&v` retourne son adresse.

Dans notre langage algorithmique, nous ne nous sommes pas préoccupé des adresses, car au niveau d'analyse où nous nous situons lorsque nous décrivons l'algorithme, l'utilisation d'adresses n'est pas nécessaire.

Cependant, le C étant un langage orienté programmation système, il manipule des adresses, et nous devons tout de même clarifier quelques notions liées aux adresses. En particulier, le langage C permet de faire de la **compilation de modules séparés**, ce qui nécessite pour être explicité, de parler des adresses.

c. Déclaration et définition

Une **déclaration** de variable indique qu'une nouvelle variable va être utilisée en précisant son nom et son type. C'est ce que nous faisons, en tête de notre programme, dans le langage algorithmique, dans la partie située entre le mot *Algorithme* et le mot *début* :

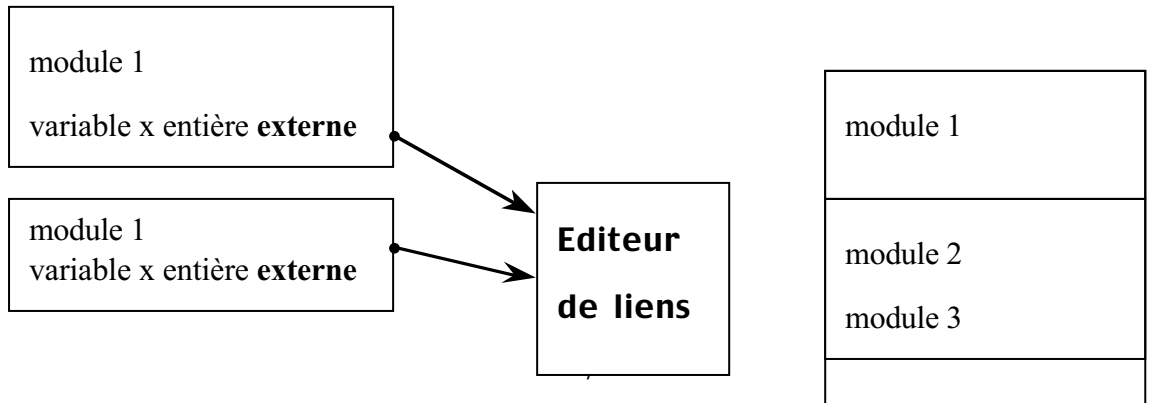
Algorithme
2 variables entières en entrée : n1, n2
1 variable entière en sortie : r
début
lecture de la première variable: n1
lecture de la seconde variable: n2
calcul du résultat : r = n1 + n2
affichage du résultat : r
fin algorithme

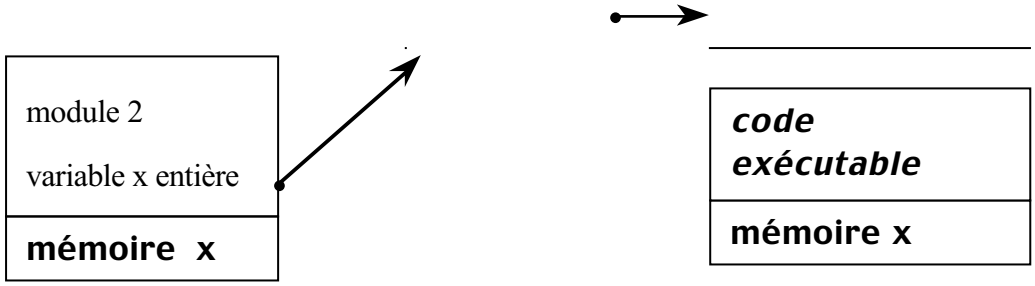
Mais dans la pratique, le compilateur doit associer à ces noms de variables une adresse en mémoire centrale. Nous avons vu que dans la première partie de ce cours, que cette association pouvait être différée jusqu'à l'édition de liens, et de manière plus générale, pour le système d'exploitation, les adresses fournies par le compilateur sont de toutes façons considérées comme relatives. Sans entrer dans les détails, il est intéressant de distinguer la notion de déclaration de variables (qui indique le type, donc les besoins en taille de stockage en mémoire) de celle de définition de variables.

Une **définition** de variable, en plus d'être une déclaration, cause l'allocation d'une zone mémoire pour cette variable (lui associe donc une adresse en mémoire centrale). Dans notre langage algorithmique, nous supposerons que nos déclarations sont des définitions.

Cette distinction définition/déclaration est à la base de la compilation séparée. A partir d'un code source (un texte de module écrit dans un langage de programmation), on peut définir un code objet (i.e. un module écrit en langage machine, mais non exécutable) faisant référence à des adresses non encore connues, pour des variables qui sont déclarées « externes » par le module concerné. Une variable peut donc être ainsi déclarée externe dans plusieurs modules, et partagée par plusieurs modules. Pour pouvoir produire un code exécutable, il faudra que cette variable soit effectivement définie dans un (et un seul) autre module.

L'**éditeur de liens** est un programme qui fait partie du logiciel de compilation et qui permet de rendre exécutable la liaison de plusieurs codes objet, en vérifiant qu'une seule adresse mémoire est bien finalement attribuée aux variables externes de même nom. Le schéma suivant explique ce principe :





d. Initialisation

Une variable étant une adresse mémoire typée, elle permet de stocker une valeur (typée). On associe une valeur à une variable ou on modifie cette valeur soit par une instruction d'**affectation**, soit par une intruction de **lecture** (la valeur provient alors de l'extérieur).

Une variable est dite **initialisée** dès lors qu'une valeur lui a été associée (par affectation, ou lecture). Rappelons que l'évaluation d'une variable consiste à y accéder et à retourner sa valeur. L'évaluation d'une variable non préalablement initialisée provoque selon les cas une erreur ou un résultat imprévisible.

Dans certains langages, dont C, on peut initialiser des variables au moment de leur définition.

Une variable initialisée est toujours définie, puisqu'elle a une valeur (donc une adresse mémoire où stocker cette valeur). Dans notre langage algorithmique, toutes les variables sont définies mais elles ne sont pas toujours initialisées.

1.5 Actions élémentaires

Une action se traduit dans un algorithme de programmation par une instruction ou un bloc d'instructions. (Un bloc d'instructions étant une suite de plusieurs instructions).

1.5.1 Affectation

L'affectation permet d'associer une valeur à une variable. Elle est, en C, de la forme :

Affectation

```
<id de variable> = <expression> ;
```

où <id de variable> est un nom de variable et renvoie à une zone de la mémoire. On distingue soigneusement dans une affectation ce qui est situé à gauche du signe d'affectation (ici, un identificateur) et ce qui est situé à droite. La partie gauche ne peut contenir ici qu'un identificateur, par contre la partie droite peut contenir une **expression**, arbitrairement complexe, désignant un calcul faisant intervenir des **constantes**, des **variables** ou des **fonctions**. On conçoit bien ce qu'est une expression numérique, mais on verra que des expressions de type booléen ou caractère ne sont pas dénuées de sens.

L'expression située à droite du signe de l'affectation est évaluée (i.e. le calcul qu'elle désigne est effectué) et la valeur retournée par cette évaluation est placée «dans» la variable, c'est-à-dire, stockée à son adresse.

Dans notre langage algorithmique, puisqu'on ne se soucie pas des adresses, on dira simplement qu'une instruction d'affectation permet d'associer une valeur à une variable. La syntaxe serait identique, en supprimant le point virgule.

NB.

- Attention à ne pas confondre le signe de l'affectation en C (=) avec le signe d'égalité qui s'écrit en C avec deux signes égal (==).
- En C, le point-virgule marque la fin d'une instruction.

Exemple

```
nbBillet = 3 ;
          /* La valeur de nbBillet est 3 */
nbBillet = nbBillet + 2 ;
          /* la valeur de nbBillet est 5 */
```

Le texte situé entre des signes /* et */ constitue un commentaire au programme et est ignoré par le compilateur du programme. Dans cet exemple, on a deux instructions d'affectation successives. Lors d'une affectation, la valeur antérieure de la variable située à gauche du signe d'affectation est ignorée (« écrasée »). L'expression située à droite du signe d'affectation est évaluée avant d'être stockée à l'adresse de la variable. Dans l'exemple ci-dessus la variable `nbBillet` dans la partie droite de l'affectation de la troisième ligne s'évalue en `3`, du fait de la première instruction d'affectation. L'expression `nbBillet + 2` est donc évaluée, le valeur calcul retourne `5,` et c'est cette nouvelle valeur qui est affectée à la variable `nbBillet` qui vaut désormais `5` et non plus `3`.

1.5.2 Lecture

L'opération de lecture permet de lire une valeur « à l'extérieur », notamment une valeur entrée par l'utilisateur au clavier. (Il existe également des opérations de lecture dans des fichiers). Quand le programme arrive à une instruction de lecture d'entrée, il s'interrompt en attendant que l'utilisateur entre une valeur. Cette valeur est alors « lue » par le programme et affectée à la variable du programme indiquée dans l'instruction de lecture d'entrée. La syntaxe de cette opération de lecture d'entrée est la suivante :

Lecture	
Algorithme	**Codage en C**
lire(<id de variable>)	`scanf(<ch de contrôle>, &<id de variable>) ;`

Dans notre langage algorithmique, la variable désignée par l'argument de *lire* prend la valeur qui sera lue en entrée. *lire* prend donc un argument (un identificateur de variable) et affecte à la variable correspondante la valeur qui sera entrée par l'utilisateur au clavier.

Dans l'opération de lecture, la valeur de la variable est modifiée : la variable n'a généralement pas la même valeur avant et après l'opération de lecture.

Exemple

Soit une variable i

i<-876
lire(i)

Si l'utilisateur tape 555, la valeur de *i* est 555. l'ancienne valeur 876 est effacée et perdue.

En C la syntaxe de l'opération de lecture est plus complexe. Nous en donnons ici une version simplifiée et nous y reviendrons plus loin. La fonction `scanf` prend deux arguments :

- Le premier argument est une chaîne de caractères, appelée « chaîne de contrôle » ou « chaîne de format », qui spécifie comment doit être interprété ce que tapera l'utilisateur ensuite. S'il s'agit de lire un entier, cette chaîne doit être "%d". Nous verrons plus loin que l'on peut lire autre chose que des entiers.
- Le deuxième argument est la variable dont la valeur doit être lue. Notons seulement que le deuxième argument du `scanf` est non pas un identificateur de variable mais une adresse de variable, ce qui est indiqué ici par l'opérateur d'acresse &.

Exemple

Traduisons l'exemple précédent en C :

```
i=876 ;
scanf("%d",&i) ;
```

Cette instruction de lecture peut être généralisée à une instruction de lecture de plusieurs variables. Dans ce cas, le programme attend de l'utilisateur qu'il fournisse autant de valeurs qu'il y a de variables à lire en argument. Il lit en entrée les valeurs dans l'ordre où elles sont annoncées, et les affecte tour à tour à chacune des variables correspondantes. La syntaxe est la suivante.

Lecture généralisée à plusieurs arguments

Algorithme	**Codage en C**
lire (<id1>, … , <idN>)	`scanf("%d…%d", &<id1>, … , &<idN>);`

Dans la syntaxe de C, c'est la chaîne de contrôle qui permet de déterminer combien de valeurs doivent être lues (ce sera le nombre de %) et le type de ces valeurs (le symbole suivant le signe %, avec ici par exemple %d qui signifie entier).

Exemple

Soit `i`, `j`, et `k`, trois variables.

i <- 876
lire(i,j,k)

donnera en langage C

```
i=876 ;
scanf(%d%d%d",&i,&j,&k) ;
```

Si l'utilisateur tape 333 444 222 au clavier. Les variables *i, j, k* prendront respectivement les valeurs 333, 444 et 222 car le caractère blanc sert de séparateur entre des nombres.

1.5.3 Ecriture

L'opération d'écriture par laquelle un programme écrit des valeurs ou les transmet à « l'extérieur » (affichage à la console, notamment) est le pendant de l'opération de lecture. Sa syntaxe est la suivante.

Ecriture

Algorithme	**Codage en C**
écrire(<expression>)	`printf(<ch de contrôle>,<expression>) ;`
écrire(<message>)	`printf(<message>) ;`

Au niveau algorithmique, on a distingué ici deux types d'opération de lecture. Dans le premier cas, l'argument est une expression (l'addition de deux variables, par exemple). Cette expression est évaluée et sa valeur est imprimée. Dans le deuxième cas, l'expression est un message et ce message est affiché *in extenso*. (En réalité, il s'agit toujours du même cas, car les messages pouvent être considérées comme des expressions constantes de type chaîne de caractères).

Exemple

Soit *i*, une variable. Considérons la séquence d'instructions suivante :

```
i<-1
écrire(i)
écrire(5+7)
écrire(«Bonjour !»)
```

```
i=1 ;
printf("%d",i) ;
printf("%d", 5+7) ;
printf("Bonjour !") ;
```

La première instruction initialise *i* à 1.
La première instruction d'écriture provoque l'affichage à l'écran de 1 (valeur associée à la variable *i*).
La seconde instruction d'écriture provoque l'affichage de 12, résultat de l'addition de 5 et 7.
Et la troisième provoque l'affichage à l'écran de Bonjour !

En réalité, ces deux types d'instructions d'écriture étant de même nature, elles peuvent naturellement se mêler : on peut mélanger des messages (chaînes de caractères à afficher telles quelles) et des expressions plus complexes à évaluer.

Exemple

Soient i et j deux variables. Considérons la séquence algorithmique suivante :

```
i<-5
j<-7
écrire(« La somme de », i, « et », j, « est », i+j))
```

Les premières instructions initialisent i et j à 5 et 7 et l'instruction d'écriture provoque l'affichage à l'écran de La somme de 5 et 7 est 12.

Le résultat est le même que si on avait :

écrire(« La somme de »)
écrire(i)
écrire(« et »)
écrire(j)
écrire(« est »)
écrire(i+j)

En revanche, on n'obtient pas le même résultat avec l'instruction suivante qui affiche à l'écran La somme de i et j est 12 :

écrire(« La somme de i et j est », i+j))

On constate sur cet exemple que ce qui figure entre guillemets est affiché sans modification alors que les autres arguments sont d'abord évalués.

En considérant qu'une chaîne entre guillemets est une expression constante qui s'évalue en elle-même, on peut donc multiplier et mélanger les deux types d'arguments au sein d'une même instruction d'écriture ayant un nombre arbitraire d'élements à imprimer (comme dans l'exemple ici avec écrire(« La somme de », i, « et », j, « est », i+j))).

Mais la syntaxe du C est un peu plus complexe de ce point de vue : c'est le premier argument de la fonction printf, la **chaîne de contrôle**, qui détermine la forme globale du message et la position des expressions à évaluer dans ce message. Les arguments à évaluer proprement dits sont donnés après la chaîne de contrôle, dans les arguments suivants. Traduisons la séquence algorithmique ci-dessus et sa variante en C (l'affichage obtenu est identique).

Exemple

```
i=5 ;
j=7 ;
printf("La somme de %d et %d est %d", i, j, i+j) ;
```

ce qui ne donne bien entendu pas le même affichage que

```
printf("La somme de i et j est %d", i+j) ;
```

La chaîne de contrôle inclut toutes les parties de message à afficher et comporte des places (notées ici `%d`) indiquant où doivent être insérées les valeurs résultant des évaluations des arguments suivants. Dans le premier cas ci-dessus, il s'agit des trois valeurs associées à i, j et à la somme de i et j. En principe, s'il y a n places réservées dans la chaîne de contrôle, il doit y avoir n arguments après la chaîne de contrôle dans le `printf`.

Remarque : nous ne manipulons à ce stade que des expressions entières mais nous verrons ultérieurement que la fonction `printf` peut manipuler toute sorte d'arguments.

Exemple : addition2

Revenons sur l'exemple de l'addition présenté plus haut.

Algorithme
2 variables entières en entrée : n1, n2
1 variable entière en sortie : r
début
lire(n1)
lire(n2)
r <- n1 + n2
écrire(« Le résultat de l'addition est « , r)
fin algorithme

Cet algorithme peut se traduire en C en le programme suivant :

```
#include <stdio.h>

int main () {
      int n1,n2,r;

      scanf("%d",&n1);
      scanf("%d",&n2);
      r=n1+n2;
      printf("le résultat de l'addition est %d",r);
      return 0;
}
```

Nous avons introduit ici en caractères gras d'autres éléments du langage C, qui sont nécessaires pour la définition correcte du programme exécutable. Considérons les pour l'instant comme des éléments purement syntaxiques. Nous reviendrons au chapitre suivant sur leur significations véritables. On peut simplement indiquer que la ligne `#include <stdio.h>` permet d'avoir accès aux fonctions `printf` et `scanf` de la bibliothèque des fonctions d'entrées/sorties, et que les autres lignes permettent de définir la fonction `main` (ou fonction principale du programme) qui spécifie l'algorithme exécuté par le programme.

Notons qu'il est toujours souhaitable de faire précéder une instruction de lecture par une **invite**, c'est-à-dire par un message qui invite l'utilisateur à taper quelque chose et qui lui précise ce que ce « quelque chose » doit être. Il est également important d'indiquer quand on passe à la ligne suivante. Pour aller à la ligne suivante après un ordre d'écriture, on utilisera en C le symbole \n qui symbolise le « caractère » *newline* ou passage à la ligne. On peut ainsi améliorer l'exemple précédent.

Exemple : addition3

Algorithme
2 variables entières en entrée : n1, n2
1 variable entière en sortie : r
début
écrire(« Entrez la première opérande : »)
lire(n1)
écrire(« Entrez la deuxième opérande : »)
lire(n2)
r <- n1 + n2
écrire(« Le résultat de l'addition est »,r)
fin algorithme

```
#include <stdio.h>

int main () {

     int n1,n2,r;

     printf("Entrez la première opérande : ");
     scanf("%d",&n1);
 printf("\n") ;     /* pour aller à la ligne */
     printf("Entrez la seconde opérande : ");
     scanf("%d",&n2);
     printf("\n") ;      /* pour aller à la ligne */
 r=n1+n2;
     printf("le résultat de l'addition est %d",r);
     return 0;
}
```

Voici ce que l'exécution de ce programme donne à l'écran (ce qui est tapé par l'utilisateur est ici en gras) :

```
Entrez la première opérande : 5
Entrez la seconde opérande :7
Le résultat de l'addition est 12
```

Remarque : on aurait pu également regrouper les instructions d'écriture successives avec

```
     printf("Entrez la première opérande : ");
     scanf("%d",&n1);
 printf("\nEntrez la seconde opérande : ");
```

1.6 Structures conditionnelles

Une structure conditionnelle permet d'introduire une instruction qui n'est exécutée que si une certaine condition est vérifiée. Cette structure a la forme suivante.

Syntaxe d'une structure conditionnelle simple

Algorithme	**Codage en C**
si (<condition>) *alors <instruction>* *finsi*	`if (<condition>)` `<instruction 1> ;`
ou	ou
si (<condition>) *alors <instruction1>* *<instruction 2>* … *<instruction N>* *finsi*	`if (<condition>) {` `<instruction 1> ;` `<instruction 2> ;` … `<instruction N> ;` `}`

L'instruction n'est exécutée que si la condition est vérifiée. Il peut s'agir d'une instruction simple (première forme) ou d'un bloc d'instructions (deuxième forme). Dans ce dernier cas l'exécution du bloc des instructions 1 à N est également subordonnée à la valeur de la condition notée entre parenthèses.

Exemple : division

Reprenons l'exemple de la division :

Algorithme
2 variables entières en entrée : n1, n2
1 variable entière en sortie : r
début
lire(n1)
lire(n2)
si (n2 ≠ 0)
alors calcul du résultat : r <- n1 / n2
écrire(r)
finsi
fin algorithme

En traduisant cet algorithme en C on obtient :

```
#include <stdio.h>

int main () {
      int n1,n2,r;

      scanf("%d",&n1);
      scanf("%d",&n2);
      if (n2 != 0) {
          r = n1 / n2;
          printf("résultat : %d",r);
      };
      return 0;
}
```

Dans ce cas, la division n'est effectuée que si le deuxième entier lu est différent de zéro.

Une structure conditionnelle peut également avoir la forme d'un branchement entre deux instructions:

Syntaxe d'une structure conditionnelle

Algorithme	**Codage en C**
si (<condition >) *alors <instruction A>* *sinon <instruction B>* *finsi*	`if (<condition>)` `<instruction A> ;` `else` `<instruction B> ;`

Cette fois, si la condition est vérifiée, c'est l'instruction A qui est exécutée et l'instruction B ignorée, et dans le cas contraire (la condition n'est pas vérifiée), l'instruction A est ignorée et l'instruction B est exécutée. La condition détermine ici un branchement alternatif du programme vers une instruction A ou une instruction B.

De même que pour la structure conditionnelle simple, les instructions A et B peuvent être remplacées par des séquences d'instructions (ou blocs d'insructions), les blocs d'instructions en C étant constitués d'une suite d'instructions figurant entre accolades.

Exemple : division2

On peut améliorer l'exemple précédent en affichant un message d'erreur pour prévenir l'utilisateur quand le diviseur est égal à 0.

Algorithme
2 variables entières en entrée : n1, n2
1 variable entière en sortie : r
début
lire(n1)
lire(n2)
si (n2 ≠ 0)
alors r <- n1 / n2
écrire(r)
sinon écrire(« erreur : pas de division par 0 »)
finsi
fin algorithme

```
#include <stdio.h>
int main () {
      int n1,n2,r;

      scanf("%d",&n1);
      scanf("%d",&n2);
      if (n2 != 0) {
        r = n1 / n2;      /* bloc de 2 instructions avec { } */
        printf("résultat : %d",r);
      }
      else                /* instruction simple */
        printf("erreur : pas de division par 0");
      return 0;
}
```

Dans ce cas, la division a lieu quand elle est définie, et si le diviseur est nul, un message d'explication est affiché pour prévenir l'utilisateur.

Nous verrons plus loin que les instructions insérées dans ces structures conditionnelles peuvent être également complexes. Signalons simplement ici que ces structures conditionnelles peuvent être imbriquées les unes dans les autres.

Exemple : division3

On peut ainsi vouloir prévenir l'utilisateur que le résultat de la division est arrondi à l'entier inférieur.

si (n2 ≠ 0)
alors r <- n1 / n2
si (n1 est divisible par n2)
alors écrire(« valeur exacte »)
sinon écrire(« valeur arrondie »)
finsi
écrire(r)
sinon écrire(« erreur : pas de division par zero »)
finsi

```
int main () {
  int n1,n2,r;
  scanf("%d",&n1);
  scanf("%d",&n2);
  if (n2 != 0) {
      r = n1 / n2;
      if (n1 == (r * n2))
          printf("Le résultat est une valeur exacte : ");
      else
          printf("Le résultat est une valeur arrondie : ");
      printf("%d",r);
  }
  else
      printf("erreur : pas de division par zero");
  return 0;
}
```

2 Structure des programmes

2.1 Forme générale d'un programme en C

Un programme manipule des valeurs stockées dans des variables qui doivent être définies. On peut définir des variables en début de programme. Ces variables sont alors qualifiées de **globales**, car elles seront définies et utilisables par toutes les fonctions définies ensuite dans le module décrit par le fichier de texte source. Il n'est cependant pas toujours nécessaire de manipuler des variables globales. Les variables peuvent en effet être introduites localement dans la définition d'un bloc. On parle alors de **variables locales**. Ces variables ne sont définies et utilisables qu'à l'intérieur du bloc d'instructions où elles ont été déclarées.

Tout programme exécutable doit contenir la définition d'une fonction appelée `main` (*main* = principale en anglais). Cette fonction joue un rôle spécial : elle est appelée au lancement du programme par une commande du sytème d'exploitation. Son résultat sera rendu à la fin de l'exécution du programme au système d'exploitation. Bien qu'elle ait un rôle spécial, sa définition est semblable du point de vue syntaxique à celle de n'importe quelle fonction.

La définition d'une fonction en C est de manière générale constituée par un **bloc d'instructions** introduit par des accolades. Un bloc d'instructions est constitué d'une suite d'instructions, mais il peut comporter une partie déclaration de variables en tête de bloc, avant la première instruction. Ces variables sont des variables locales, définies uniquement pendant l'exécution du bloc définissant la fonction.

Remarque : Il ne peut figurer dans cette partie « déclarations de variables locales », que des déclarations de variables, et pas de définition de fonctions (contrairement à la partie déclarations globales, située en tête de programme). Nous reviendrons plus loin sur cette restriction du C.

Exemple

```
#include <stdio.h>
      /* Partie déclarations (globales) */
      /* (types, variables, fonctions etc.) */

int main() {
      /* Partie déclarations de variables (locales) */
      int x;

      /* Partie actions (suite d'instructions) */
      printf("Saisir une valeur entière :");  /* (1) */
      scanf("%d",&x);                         /* (2) */
  printf("\n") ;         /* (3) */
      printf("Le double est : %d", 2*x);      /* (4) */
      return 0 ;
}
```

La suite ordonnée et finie d'instructions constituant le programme est la partie actions de la fonction `main`, qui commence à la première instruction du bloc (notée ici notée (1) dans le commentaire). La ligne qui précède est une déclaration de variable locale (x, à valeur entière).

L'exécution du programme commence à la première instruction de la fonction `main`, à savoir l'instruction (1). Remarquez que toute instruction élémentaire se termine par un point virgule qui fait partie de l'instruction. Les commentaires, qui ne constituent pas des instructions exécutables mais ont pour objectif d'améliorer la compréhension, sont délimités par les caractères /* et */ et ignorés dans l'interprétation du programme.

2.1.1 Définition de la fonction main

La définition d'une fonction comporte toujours le type de la valeur retournée par la fonction et la déclaration du type de ses paramètres. La déclaration des types des paramètres figure entre parenthèses, après le nom de la fonction. Ici, dans le cas de cette fonction `main`, il n'y en a pas, ce qui fait qu'on l'on a simplement une parenthèse ouvrante et une parenthèse fermante. (Nous reviendrons plus tard sur la définition des paramètres d'une fonction.) Par contre, le type de la valeur retournée par une fonction doit toujours être déclaré. Ici, il est de type `int`, et il apparaît au niveau de la définition de la fonction avant le nom de la fonction :

```
int main()
```

Cette déclaration est celle du type de la valeur rapportée par le calcul de la fonction, i.e. la valeur de « retour » ; ce type doit coïncider avec celui de la valeur effectivement retournée par la fonction dans la dernière instruction exécutée du bloc. Cette instruction en C doit nécessairement être une instruction de la forme `return <expression>`, indiquant que la fonction est terminée et définissant le résultat du calcul sous forme d'une expression.

Exemple

Le programme qui suit contient trois blocs d'instructions, certains inclus dans d'autres. Les instructions (7) à (17) constituent le bloc de définition de la fonction `main`. Les instructions (10) à (12) constituent également un bloc d'instructions au même titre que le bloc constitué des instructions (15) et (16). Ces deux blocs alternatifs se terminent par une instruction `return` :

```
(1)  #include <stdio.h>
(2)
(3)  int main () {
(4)
(5)      int n1,n2,r;
(6)
(7)      scanf("%d",&n1);
(8)      scanf("%d",&n2);
(9)      if (n2 != 0) {
(10)         r = n1 / n2;
(11)         printf("résultat : %d",r);
(12)         return 0 ;
(13)     }
(14)     else {
(15)         printf("erreur : pas de division par 0");
(16)         return 1 ;
(17)     }
(18) }
```

Le langage C ne comporte aucune règle de mise en page. On peut écrire une instruction n'importe où sur une ligne, on peut même en disposer plusieurs sur une même ligne, mais il est recommandé de n'écrire au plus qu'une instruction par ligne. On peut également écrire une même instruction sur plusieurs lignes. Une exception cependant, à l'intérieur d'une chaîne de caractères entre guillemets, il ne faut pas passer à la ligne suivante, sauf à précéder le passage à la ligne du caractère \ pour que le compilateur ignore ce passage à la ligne. L'exemple qui suit illustre ce point.

Il est néanmoins très important d'écrire les programmes de façon à les rendre facile à lire et à comprendre. Nous attachons beaucoup d'importance à la mise en page des programmes, et en particulier, à la manière dont on aligne ensemble les instructions faisant partie d'un même bloc. Le fait d'aligner verticalement le début des lignes en suivant un certain alinéa à l'intérieur d'un même bloc s'appelle **l'indentation**. On peut suivre des règles d'indentation relativement strictes. Cela favorise la lisibilité des programmes car la notion de bloc est très importante. Les compilateurs ne tiennent pas compte de l'indentation, mais le programmeur est libre de faire la mise en page qu'il souhaite, et il suffit de comparer les deux présentations suivantes du même programme pour se rendre compte de l'intérêt d'une bonne mise en page :

```
#include <stdio.h>
int main() { int x;
printf("Saisir un \
entier :");
scanf("%d",&x);printf
("Le double est :%d",
2*x);return 0;}
```

```
#include <stdio.h>

int main() {
   int x;

   printf("Saisir un entier :");
   scanf("%d",&x);
   printf("Le double est: %d", 2*x);
   return 0 ;
}
```

Hormis les règles générales d'indentation, **une règle importante de mise en page consiste à laisser au moins une ligne blanche entre la partie déclarations de variables d'un bloc et la partie actions.**

2.1.2 La partie déclarations (globales)

Dans la partie déclarations en tête de fichier, le programmeur liste les objets spécifiques qui seront manipulés par son module, à savoir des constantes, des variables, des types, des procédures et des fonctions. Pour le moment, on ne s'intéresse qu'aux déclarations de constantes et de variables, mais ce sera bien sûr plus tard le lieu de définition (ou de déclarations) des fonctions.

Les variables, comme les constantes, sont des données typées. Nous allons maintenant définir plus clairement ce qu'est un type.

2.2 Les types

Définition : Un type est un ensemble de valeurs et pour chaque type il existe un ensemble d'opérations pouvant être appliquées sur ces valeurs.

2.2.1 Les types couramment rencontrés dans les langages impératifs

Dans les langages impératifs, les types les plus couramment rencontrés sont les suivants :

2.2.2 Le type entier

Le type *entier* désigne un ensemble fini d'entiers.

2.2.3 Le type réel

Le type *réel* désigne un ensemble fini de nombres réels.

2.2.4 Le type booléen

Le type *booléen* désigne deux valeurs possibles {vrai, faux} qui sont celles de la logique mathématique.

2.2.5 Le type caractère

Il s'agit d'un ensemble de caractères comprenant notamment les caractères alphabétiques latins 'a', 'b', 'B'…, les chiffres '0', '1'..., les signes de ponctuations, l'espace et divers autres caractères spéciaux. Dans le langage algorithmique, on prendra soin de noter les valeurs de type *caractère* entre apostrophes (*quotes*) pour les distinguer des valeurs numériques.

2.2.6 Le type chaîne de caractères

Une valeur de type *chaîne de caractères* est une suite ordonnée de valeurs de type *caractères*.

2.2.7 Les types de base du langage C

En C, les types de bases sont plus limités.

2.2.8 *Le type* `char`

Une valeur de type char est codée sur 1 octet. L'ensemble de caractères utilisé est celui défini pour l'ordinateur sur lequel tourne le compilateur (l'ensemble de caractères le plus couramment employé est l'ASCII).

En C, un caractère se note entre guillemets simples (*quotes*) mais on peut également utiliser directement le code ASCII octal du caractère en le précédant du caractère `\`. Ainsi `'a'` et `'\141'` désignent le même caractère.

Remarques : On notera que `'a'` et `'A'` désignent des caractères différents, correspondants respectivement aux codes ASCII `97` et `65`. Il convient également de bien distinguer le caractère `'4'` et l'entier `4`.

Mentionnons également les caractères spéciaux suivants :

`'\n'` Passage à la ligne

`'\t'` Tabulation

`'\a'` Bip sonore

En C, à toute valeur de type caractère correspond une valeur entière qui est son code ASCII.

Exemple :

```
#include <stdio.h>

int main() {
 char c;

 c='a' ;
 printf("Le code ASCII \ndu caractere %c est : %d", c, c) ;
 return 0 ;
}
```

Si ce programme est exécuté, on aura l'affichage suivant :

Le code ASCII
du caractère a est : 97.

2.2.9 *Le type* `int`

Une valeur de type `int` est codée sur 2 ou 4 octets (cela dépend des machines). Une valeur de type `int` peut être fournie soit en notation décimale (comme 157 ou +157 ou encore –4530), le premier chiffre n'étant pas 0 ; soit en code octal si le premier chiffre est zéro (exemple 0142) ; soit en code hexadécimal si la suite de chiffres commence par 0x ou 0X (exemple 0XA4B).

2.2.10 *Le type* `float`

Une valeur de type `float` est codée sur 4 octets. Le nom `float` provient de nombres à virgule flottante pour désigner le codage le plus couramment utilisé pour stocker en machine un nombre réel. Dans ce codage, le nombre de chiffres après la virgule n'est pas constant.

Il existe deux notations possibles d'une valeur de type float :

- La notation décimale : [+ ou -]m.n où m et n sont des valeurs de type int quelconque, comme +56.87 ou –45.0 ou 0.765
- La notation scientifique (mantisse et exposant) : [+ ou -]m.n e[+ ou -]exp où m, n et exp sont des valeurs de type int.

Exemple : $1.2e^2 = 1.2 * 10^2 = 120.0$

2.2.11 *Le type* `double`

Une valeur de type `double` est un nombre à virgule flottante ; elle est codée sur 8 octets.

Remarque 1:

Il n'existe pas en C de type booléen. Toutes les expressions dont la valeur est une valeur logique (négation d'une expression, conjonction ou disjonction d'expressions) sont considérées comme fausses si elles sont nulles, toutes les autres valeurs étant considérées comme vraies.

Remarque 2 :

Il n'existe pas non plus de type de base chaîne de caractères (en fait il en existe un, le type `String`, mais pour l'utiliser il faut faire appel à une bibliothèque et nous n'aborderons cette possibilité que plus tard). Néanmoins, si on ne peut pas actuellement utiliser une variable de type chaîne de caractères en C, on peut d'ores et déjà manipuler une valeur de type chaîne de caractères notée entre guillemets.

2.2.12 Les qualificatifs appliqués aux types de base

Chaque type de base peut être précédé d'un qualificatif permettant de limiter ou d'étendre la précision du codage.

Qualificatifs appliqués au type `int` : `signed` ou `unsigned`

Il est possible d'appliquer les qualificatifs `signed` ou `unsigned` au type `int`.

Les valeurs entières relatives sont appelées des valeurs signées : avec un codage en complément à deux d'une valeur entière relative, le bit de poids fort est utilisé pour coder le signe. Ainsi, une valeur de type `signed int` codé sur 2 octets permet de coder des valeurs entières relatives variant entre : -2^{15} et $+2^{15}-1$. Par défaut, le type `int` est signé.

Les valeurs entières naturelles sont des valeurs sans signe (`unsigned`) et toujours positives. Le bit de poids fort n'est alors plus réservé au codage du signe. Ainsi, une valeur de type `unsigned int` codée sur 2 octets permet de coder des valeurs entières allant jusqu'à $2^{16}-1$.

Selon que le qualificatif est `signed` ou `unsigned`, les valeurs de type `int` varient dans les intervalles suivants :

	`signed`	`unsigned`
`int`	$[-2^{15} , +2^{15}-1]$ ou $[-2^{31} , +2^{31}-1]$	$[0 , +2^{16}-1]$ ou $[0 , +2^{32}-1]$

Qualificatifs appliqués au type `int` : `short` ou `long`

Il est également possible d'appliquer les qualificatifs `short` ou `long` au type `int`. Un `short int` (ou `short`) est codé sur 2 octets. Un `long int` (ou `long`) est codé sur 4 octets. Ainsi, quelle que soit la machine, on a la relation : `short` $\subseteq$ `int` $\subseteq$ `long`. Selon que le qualificatif est `short` ou `long`, les valeurs de type `int` varient dans les intervalles suivants :

	`short` (2 octets)	`long` (4 octets)
`int`	$[-2^{15} , +2^{15}-1]$	$[-2^{31} , +2^{31}-1]$

On peut appliquer les qualificatif `unsigned` et `signed` à une valeur de type `short` ou `long`.

Qualificatif appliqué au type **`double`** : ***`long`***

Une valeur de type `long double` est une valeur à virgule flottante codée sur 12 octets.

2.2.13 Lecture et écriture de valeurs de différents types

Le tableau ci-dessous précise les spécificateurs de type à utiliser dans la chaîne de format d'une fonction `printf` ou `scanf` pour lire ou écrire une valeur d'un type prédéfini quelconque :

Spécificateur	type	Spécificateur	type
`%d`	`int`	`%lu`	`unsigned long`
`%hd`	`short`	`%c`	`char`
`%ld`	`long`	`%f`	`float`
`%u`	`unsigned int`	`%lf`	`double`
`%hu`	`unsigned short`	`%Lf`	`long double`

Exemple

```
#include <stdio.h>
/* Ce programme a pour objet de diviser deux valeurs reelles
saisies au clavier par un utilisateur */
int main () {
      double r1,r2 ;
      double resu;

      printf("entrez deux reels : ");
      scanf("%lf%lf", &r1, &r2);
      if (r2 != 0) {
        resu = r1 / r2;
        printf("\nDivision : %lf", resu);
      } ;
  return 0 ;
}
```

Remarque : `scanf` permet ici de lire deux valeurs de type `double`. Il faut séparer ces valeurs par un caractère d'espacement (espace, tabulation, ou nouvelle ligne).
Si à l'exécution l'utilisateur saisit les valeurs 4,5 et 3,215, l'affichage à l'écran sera :
entrez deux reels : **4.5 3.215**
Division : 1.399689

2.3 Déclaration et définition de variables

On a vu qu'en C, une variable peut être déclarée en tête d'un bloc d'instructions, avant la première instruction exécutable du bloc. De manière plus générale, la déclaration d'une variable ne peut se faire qu'en des points particuliers d'un module de programme contenu dans un fichier. Une variable peut être déclarée globale au programme, et dans ce cas, elle sera

accessible par tous les modules. Ou bien une variable peut-être locale, et elle n'est connue qu'à l'intérieur d'un module ou d'un bloc.

En C, une variable est globale au module décrit par le fichier source d'un module si elle figure dans la partie déclarations globales d'un fichier, ou bien elle peut être déclarée locale, en tête de définition d'un bloc particulier d'instructions - que ce soit un bloc d'instructions contenu dans une instruction conditionnelle par exemple, ou un bloc de définition de fonction, comme par exemple, celui de la fonction `main`.

Pour déclarer une variable d'un certain type, on utilise des identificateurs de types. Nous avons déjà rencontré des identificateurs de types prédéfinis : int, char, float etc. La syntaxe pour déclarer une variable de type `<id de type>` est la suivante :

```
<id de type> <id de variable>;
```

La déclaration d'une variable introduit un identificateur (nom) de variable qui a dès lors un usage réservé. Elle associe également un (et un seul) type à l'identificateur. Toutes les valeurs pouvant être prises par la variable doivent être compatibles avec son type. Par exemple, attribuer à une variable de type `unsigned short` une valeur inférieure à 0 ou supérieure à 2^{16}-1 provoque un résultat imprévisible à l'exécution car la valeur attribuée serait d'abord arbitrairement tronquée.

Il est possible de définir simultanément un certain nombre de variables de même type comme suit :

```
<id de type> <id de variable1>, <id de variable2>,...;
```

On distingue cependant on l'a vu la **déclaration** d'une variable et sa **définition**. La déclaration des variables spécifie pour le compilateur le type associé à chaque identificateur ; elle ne réserve cependant pas toujours l'espace mémoire nécessaire au stockage des données associées à l'identificateur. Les déclarations qui allouent aussi un espace mémoire sont appelées des **définitions**. Les exemples que nous avons donnés jusqu'à présent étaient tous des exemples de définitions.

La définition de variables indique au compilateur la place mémoire qu'il faut attribuer à chaque variable. Le compilateur gère une table des symboles et réserve la place nécessaire à une adresse particulière.

Mais une déclaration globale peut être précédée du qualificatif `extern`, qui spécifie que la mémoire pour l'objet ainsi déclaré a été réservée ailleurs, c'est-à-dire dans le code d'un autre module. La déclaration du type est néanmoins utilisée par le compilateur pour vérifier la cohérence des types, mais la liaison effective entre le nom d'une variable externe et son adresse mémoire sera réalisée plus tard, par l'éditeur de liens, qui harmonise les déclarations faites entre les différents modules. Une variable ne doit être **définie** que dans un seul fichier, mais elle peut être **déclarée externe** dans plusieurs fichiers.

Une gestion cohérente des déclarations peut être facilitée en utilisant des fichiers appelés fichiers d'en-têtes, qui contiennent des déclarations de variables externes utilisées par plusieurs modules. Nous reviendrons plus tard sur ces questions.

2.4 Initialisation

On a distingué également la définition d'une variable et son **initialisation**. L'initialisation consiste à donner une première valeur (valeur *initiale*) à la variable. Il est possible d'initialiser une variable lors de sa définition de la façon suivante :

```
<id type> <id variable1> = <valeur>, <id variable2>, ...;
```

Une variable définie dans un bloc d'instructions n'existe que dans ce bloc et dans les blocs qu'il contient. Ainsi on a :

```
#include <stdio.h>
int main() {
  /* Partie déclaration de variables */
  float x;
  int e;
  /* x et e peuvent être utilisées */
  …
     {
     int a,b;
     /* x, e, a et b peuvent être utilisées ! */
     …
          {
          char v ;
          /* x, e, a, b et v peuvent être utilisées ! */
               …
          } /* v n'est plus définie */
     /* x, e, a et b peuvent être utilisées ! */
     …
     } /* a et b ne sont plus définies */
 /* seules x et e peuvent être utilisées */
 …
} /* Fin du programme et x et e n'existent plus */
```

2.5 La définition de constantes

Dans un programme on peut avoir besoin de valeurs qui ne varient pas durant toute l'exécution du programme. Par exemple, on sait qu'une page de texte à 60 lignes, que π vaut 3.14159, etc. Des valeurs qui ne varient pas sont appelées des *valeurs constantes*.

On peut utiliser de telles valeurs directement dans le corps d'un programme (sans avoir à les déclarer au préalable). Par exemple dans le programme qui calcule et affiche le double d'une valeur entière, la valeur 2 est une valeur constante. Mais, si on souhaite changer une valeur constante, comme la largeur d'une page par exemple, il faut modifier toutes les lignes du programme où cette valeur intervient. Il est alors très facile de faire des erreurs !

Pour faciliter la mise à jour des programmes ainsi que leur lisibilité, on a la possibilité en C d'associer un identificateur à une valeur constante. Un identificateur qui représente une valeur constante est appelé une *constante*. Une constante peut être définie en n'importe quel point d'un programme mais il est fortement conseillé de le faire **en début de fichier**. La syntaxe est la suivante :

```
#define <identificateur> <texte substitutif>
```

Attention : une définition introduite par `#define` ne se termine pas par un point virgule.

Ces définitions de constantes éclairent non seulement la lisibilité des programmes mais permettent aussi d'améliorer leur maintenance et leur portabilité.

Exemple

```
#include <stdio.h>
#define FACTEUR 2
#define EXIT_SUCCESS 0

int main() {
                    /* Partie déclarations de variables */
        int x;
                    /* Partie action */
        printf("Saisir une valeur entière :") ;
        scanf("%d",&x);
        printf("\nLe resultat de la multiplication par %d est :\
%d", FACTEUR, FACTEUR*x) ;
        return EXIT_SUCCESS ;
}
```

FACTEUR est ici synonyme de 2. Il ne s'agit pas d'une variable qui permet de référencer une valeur. C'est en réalité une première phase de compilation (le préprocesseur) qui remplace toutes les occurrences de FACTEUR (qui apparaissent dans le texte du programme) par 2, avant de passer le nouveau texte au compilateur proprement dit. Un appel à une commande du préprocesseur est indiqué ici par un caractère # en tête de ligne.

En réalité, la commande `#define` permet de définir plus que des constantes, et ici FACTEUR pourrait être remplacé par n'importe quel texte, comme par exemple, par 2*x + 1, ce qui donnerait finalement pour la compilation de l'instruction d'écriture :

```
        printf("\nLe resultat de la multiplication par %d est :\
```

%d", **2*x + 1**, **2*x + 1***x) ;

Nous avons défini ici la constante EXIT_SUCCESS, mais elle est en réalité définie dans la librairie standard. (Cela permet ici de rendre le programme plus portable, car si, dans la plupart des systèmes d'exploitation, une terminaison correcte d'un programme est encodé par une valeur de retour 0, sous système VMS, c'est la valeur 1 qui signale le succès.)

2.6 Les expressions

2.6.1 Définition d'une expression

Une expression est une formule spécifiant un calcul combinant des variables, des constantes, des valeurs littérales, des opérateurs ou/et des fonctions. Le rôle d'une expression est d'obtenir un résultat (final ou intermédiaire), que l'on appelle sa valeur. Le type du résultat est considéré comme le type de l'expression. Une expression peut être évaluée en un instant particulier de l'exécution du programme, cette évaluation renvoie un résultat unique et typé.

Terminologie : une expression se compose d'**opérateurs**, d'**opérandes** et de **délimiteurs**. Les délimiteurs vont toujours par paire, un ouvrant et un fermant. Ainsi, dans l'expression :

```
a * (3 + b)
```

il y a deux opérateurs : '*' et '+' ; trois opérandes : 'a', '3', et 'b' ; et deux délimiteurs : '(', et ')'.

2.6.2 Opérateurs courants dans les langages impératifs

Les opérateurs ont un ou deux arguments ; dans le premier cas, il s'agit d'opérateurs unaires et dans le second d'opérateurs binaires. La *signature* (parfois appelé aussi type) d'un opérateur spécifie le type des expressions qu'il accepte en arguments ainsi que le type de la valeur renvoyée comme résultat du calcul :

`signature : type argument1 × type argument2 → type résultat`

- Les opérateurs de comparaison :

Opérateurs	<	≤	=	>	≥	≠
En C	<	<=	==	>	>=	!=

Ce sont des opérateurs binaires dont le résultat est booléen et dont les arguments sont des nombres ou des caractères. Leur signature est la suivante :

```
Entier          entier
Réel        ×   réel        → booléen
Caractère       caractère
Chaîne          chaîne
```

Une expression mettant en jeu l'un de ces opérateurs est appelée une expression booléenne. Une **condition** est une expression booléenne dont l'évaluation conditionne un branchement conditionnel.

- L'opérateur numérique unaire « moins » : `-`

La signature de cet opérateur est :

`entier → entier`

ou

`réel → réel`

- Les opérateurs numériques binaires courants :

Opérateurs	+	-	*	Div entière	Div réelle	modulo
En C	+	-	*	/		%

Les arguments de l'opérateur modulo (%) doivent être de type entier. Si les arguments de l'opérateur / sont de type entier c'est la division entière qui est réalisée ; si l'un au moins des deux opérateurs est de type réel, c'est division réelle qui est réalisée.

Les arguments et résultats de ces opérateurs sont numériques. Le type des arguments définit le type des résultats. La signature de ces opérateurs est :

`entier × entier → entier`

ou

`réel × réel → réel`

ou encore

`entier` × `réel` → `réel`

- L'opérateur booléen unaire : ¬ en logique

L'opérateur unaire de négation (`!` en C) rend la valeur booléenne inverse de celle de son argument. Sa signature est :

`booléen` → `booléen`

Il est défini par la table de vérité suivante :

A	¬A
vrai	faux
faux	Vrai

- Les opérateurs booléens binaires : Λ et V

Il s'agit des opérateurs « et » (&& en C), « ou » (|| en C). Leur signature est la suivante :
`booléen` × `booléen` → `booléen`

Ces opérateurs se définissent par les tables de vérité suivantes :

Λ

A \ B	Vrai	Faux
Vrai	Vrai	Faux
Faux	Faux	Faux

V

A \ B	vrai	Faux
vrai	vrai	Vrai
faux	vrai	Faux

2.6.3 Evaluation d'une expression

C'est l'évaluation de l'expression qui lui fait retourner une valeur. Ainsi, à supposer que...

- la constante `pi` a la valeur `3.1415`,
- les variables réelles `r1`, `r2`, `r3` ont respectivement les valeurs `3.3533`, `0.1415` et `0.2118`,
- la variable `e` a la valeur entière `12`,
- la variable `c` désigne le caractère `'a'.`

... toutes les expressions suivantes sont des expressions réelles de valeur `3.1415` :

- `3.1415`
- `pi`,
- `r1-r3`,
- `e/4+r2`,
- `(40/e)+r2`,

... et toutes les expressions suivantes sont des expressions booléennes de valeur faux :

- `pi>r1`,
- `c=='c'`,
- `((40%e)>5) || (!e).`

Si ultérieurement, au cours de l'exécution du programme, la variable `e` est modifiée et prend la valeur `25`, alors l'expression `((40%e)>5) || (!e)` prend la valeur vrai.

L'évaluation d'une expression dépend :

- de l'ordre de priorité entre les opérateurs

Priorité	Opérateur
1	& !
2	* / %
3	+ -
4	< <= > >=
5	== !=
6	&&
7	‖

- des parenthèses qui permettent de modifier les priorités,
- à priorité équivalente, l'évaluation se fait de gauche à droite.

En règle générale, il est prudent d'utiliser des parenthèses.

2.6.4 Les conversions de type

a. Conversion automatique

Quand un opérateur à des opérandes de types différents, ils sont convertis en un type commun : c'est toujours le type de plus grande précision qui l'« emporte ».

Attention : Dans une instruction d'affectation, le type de l'opérande de droite est converti en type de l'opérande de gauche.

Exemple :

```
int x;
float y;

x = y;  /* la valeur de type float de y est automatiquement
           convertie en int avant d'être
           affectée à la variable x de type int */
```

b. Conversion forcée

On peut forcer la conversion du type de la valeur retournée par une expression (on parle de **cast** en anglais) en précédant l'expression du type forcé entre parenthèses.

Exemple :

```
int x;
```

```
float y=8.5;
x = ((int) y )% 4;
```

L'opération de *cast* sur y permet d'appliquer l'opérateur % (défini sur les entiers) à un réel.

Le type de `y` est float mais l'expression `(int) y` a pour valeur la partie entière de `y`. C'est cette valeur entière qui sera le premier opérande pour l'opérateur modulo.

Bibliographie

[1] *Le langage C, norme ANSI*, Kernigham & Ritchie, 2e ed., Dunod.

[2] *Belle programmation et langage C*, Yves Noyelle, cours de l'école Supérieur d'Electricité dans la collection TECHNOSUP, Ellipses Editions, 2001.

[3] *Initiation à la programmation*, C. Delannoy, Editions Eyrolles.