Université Paris 13
Institut Galilée
Licence 1ère année
2006-2007

Programmation Impérative
Polycopié de cours n° 1

C. Recanati

L.I.P.N.

http : //www-lipn.univ-paris13.fr/~recanati

Table des matières

1 Premières notions de programmation

1.1 Généralités

La programmation recouvre l'ensemble des techniques permettant de résoudre des problèmes à l'aide de programmes s'exécutant sur un ordinateur. Une étape essentielle consiste en l'écriture d'un texte de programme dans un langage particulier (un langage de programmation). Mais l'écriture du programme, bien que fondamentale, n'est qu'une étape du processus de programmation que l'on peut décomposer de la manière suivante :

1. **L'analyse et la spécification** du problème, qui permettent de préciser 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, ainsi que les relations exactes entre les données et les résultats.

2. **La conception (ou modélisation)** : il s'agit généralement de déterminer la méthode de résolution du problème (un **algorithme**) et d'en identifier les principales étapes. C'est également dans cette phase que l'on conçoit la manière dont on va mettre en œuvre les fonctionnalités d'un logiciel (celles qui ont été identifiées dans la phase d'analyse).

3. **L'implantation** (ou codage) dans un ou plusieurs langages de programmation particuliers. Il s'agit de traduire la méthode et les algorithmes préconisés en un ou plusieurs textes de programmes. 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 mots et des règles d'écriture (la « grammaire ») qui détermine la structure d'un texte de programme correct, c'est-à-dire, que la machine peut transformer en code exécutable grâce à un **compilateur** prévu pour ce langage de programmation. Cette syntaxe doit être rigoureusement suivie. Un seul mot ou caractère mal placé et le texte du programme ne peut être traduit en code exécutable. Franchir cette étape n'est pas très difficile car on bénéficie de l'aide du compilateur qui signale une à une les erreurs qu'il rencontre. Mais on peut malheureusement aussi commettre des erreurs de type **sémantique** (erreurs relatives au « sens » du programme) : dans ce cas, le texte du programme peut être transformé en code exécutable par le compilateur, mais ce code peut provoquer des interruptions brutales avec certaines données, ou pire, fournir des résultats erronés.

Le risque d'erreurs de codage étant en fait relativement élevé, l'implantation doit toujours être suivie d'une phase de mise au point dans laquelle on effectue des tests permettant de vérifier la robustesse du programme obtenu. Le programme doit également encore ensuite être corrigé pour être simplement maintenu à son niveau de performances dans un environnement en constante évolution - ce qui fait que la phase de mise au point est de durée a priori indéterminée.

Notons que l'écriture du programme proprement dit ne vient qu'en troisième étape du processus de programmation, et que les deux premières étapes sont souvent longues. Il faut aussi être conscient du fait qu'il y a un travail considérable entre la fin de la seconde étape et l'entrée du texte du programme dans l'ordinateur. En particulier, on aura parfois à se poser des questions d'architecture de programmes pour faire fonctionner ensemble des parties développées avec des éléments préexistants (récupération de données dans des bases de données ou sur Internet, etc.), ou pour articuler des modules de programmes écrits dans des langages de programmation différents.

De même, la phase de tests et de mise au point d'un programme reste souvent négligée par les débutants en programmation, mais les problèmes apparaissant alors peuvent être fort complexes et coûteux de sorte que si cette étape n'est pas réellement menée à bien, il est souvent plus simple de repartir de zéro, même dans le cas de programmes industriels ayant coûté plusieurs hommes/années.

La qualité du travail accompli dans les deux premières étapes reste bien entendu primordiale pour la qualité du programme produit. Différentes branches de compétences et de techniques basées sur des modélisations mathématiques se sont d'ailleurs développées avec succès pour résoudre des problèmes de structuration de données, donnant ainsi naissance à une branche de l'informatique : **l'algorithmique**. (On en verra une introduction au second semestre via la présentation de quelques algorithmes de tri et de recherche d'élements stockés dans des structures de données classiques).

1.2 Programmation impérative

Les structures de données et la nature des algorithmes étudiés dans le cadre classique de l'algorithmique sont intimement liées à la nature des langages de programmation utilisés. Ces langages dits « **impératifs** », sont basés sur la notion d'action exécutée, l'action la plus typique étant **l'affectation** de variable. Le concept de variable dans ces langages de programmation est très différent de la notion mathématique. Grossièrement, une variable est une case mémoire de l'ordinateur dans laquelle on peut stocker une valeur de donnée, numérique ou autre. L'opération d'attribution de valeur à une variable s'appelle l'affectation. La valeur d'une variable peut être récupérée et modifiée à volonté, et la programmation dans ces langages est basée principalement sur l'utilisation et la modification judicieuse de variables. Ces langages sont qualifiés d' impératifs, car un programme consiste en une série d'ordres (ou instructions) donnés à exécuter par la machine. Parmi ces ordres, l'affectation est une opération fondamentale.

Le langage C que nous allons utiliser dans ce cours est un représentant typique des langages impératifs. Ce type de langages est particulièrement adapté à la structure des ordinateurs classiques, car, bien que qualifiés de langages de programmation « évolués » - par opposition aux langages machine[1] - ils restent très proches dans leur conception de ces langages de bas niveaux. C'est une des raisons qui les rend incontournables, car cette proximité de structure leur assure une supériorité de performances indéniable dans le contexte des architectures de machines actuelles. Une autre raison importante est que, ayant été historiquement inventé les premiers, la plupart des langages utilisés aujourd'hui (comme les langages à objet), comportent eux aussi un noyau « impératif » et concervent la possibilité d'écrire des modules de programmes dans ce style de programmation.

D'autres types de langages s'écartant plus de cette structure existent, comme les langages dits « fonctionnels » ou « déclaratifs ». Ces langages permettent de résoudre et spécifier plus facilement certains problèmes, en offrant d'autres moyen de structurations de données (et en ne manipulant pas que des nombres ou des caractères). Mais nous n'aurons guère le temps d'aborder cette année la programmation qu'ils mettent en oeuvre.

---

[1] C'est le code en langage machine, généré après compilation du programme, qui est interprété au moment de l'exécution du programme. Le but d'un langage de programmation est d'atténuer l'écart entre la séquence d'actions qu'il décrit, et celle déclenchée au niveau électronique par l'exécution du programme en langage machine.

Bien qu'un cours de programmation ne puisse se réduire à un cours sur un ou des langages de programmation (cf. les premières étapes d'un processus de programmation), leur apprentissage et leur pratique restent une nécessité absolue, car c'est avec l'écriture effective de programmes et leur passage en machine que le processus de programmation peut finalement aboutir à la résolution de problèmes. Il faut donc bien avoir malgré tout une petite expérience et une certaine idée de ce en quoi consistera le programme, pour franchir correctement les premières étapes d'analyse et de spécification.

Nous allons donc au cours de ce premier semestre, nous consacrer d'abord à l'apprentissage et à la pratique d'un langage impératif, le langage C, en nous exerçant sur des problèmes simples pour lesquels les deux premières phases ne sont pas trop problématiques.

Le langage C souffre de la réputation d'être un langage de bas niveau, car il manipule le même type d'objets que la plupart des machines, c'est-à-dire des caractères, des nombres et des adresses d'emplacement mémoire. Il n'a guère d'opérations qui manipulent directement des chaînes de caractères, ou qui simplifient les accès en lecture/écriture sur des fichiers, ou encore permettant de synchroniser des opérations en parallèle. Mais sa taille modeste lui assure finalement l'avantage de l'accessibilité, et un programmeur peut raisonnablement espérer connaître et comprendre assez rapidement tout le langage et l'utiliser régulièrement.

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 dans l'industrie, car c'est un langage généraliste et il reste indépendant de toute architecture de machine particulière.

Son apprentissage permet en outre d'apprendre ensuite facilement d'autres langages du même type (Pascal, Fortran, Cobol, Basic), et un même d'autres types de langages, comme des langages à objet, car un certain nombre d'entre eux en dérivent (C++, Perl, ou Java) et présentent avec lui des similitudes syntaxiques.

Au second semestre, nous complèterons ce premier apprentissage de la programmation par deux volets d'approfondissement : un premier volet introduira quelques premières notions d'algorithmique, base théorique de la programmation impérative, et un deuxième volet permettra de comprendre la manière dont les programmes peuvent finalement être exécutés, en précisant quelques particularités de l'architecture des machines actuelles.

1.3 Variables : identificateur, valeur et type, adresse

Une **variable** est une zone de la mémoire centrale à laquelle on donne un nom, ou **identificateur**, qui permet de la désigner dans le programme, et qui contient une **valeur**. Cette valeur reste stockée dans cette zone et ne change que si l'on **affecte** une autre valeur à la variable.

Une variable ne contient qu'une seule valeur à la fois, et cette valeur est la dernière qu'on lui a affectée. Tant qu'une première valeur (valeur initiale) n'a pas été affectée à une variable, la valeur de cette variable est (a priori) indéterminée. Il est souvent important de préciser la valeur initiale des variables, et l'on appelle cette première opération d'affectation **l'initialisation**.

1.3.1 Identificateur, valeur et type

Le codage des valeurs en machine consiste en une suite de 0 et de 1. Pour pouvoir déterminer la valeur correspondant à une telle représentation, il est nécessaire de connaître le **type** de

cette valeur (le codage s'en déduit et la valeur est alors déterminée, mais le type ne fait pas partie du codage). Ainsi par exemple, le code 01001111 représente aussi bien l'entier 79 (codage d'un entier sur 8 bits en binaire) que le caractère 'O' (un caractère est codé ici par un entier : son code Ascii).

A chaque variable est donc associé un **type** qui indique l'ensemble abstrait des valeurs qu'elle peut prendre, et qui permet de récupérer sa valeur. Le type d'une variable détermine également les opérations que l'on peut effectuer avec elle. Il existe plusieurs types classiques dans les langages de programmation : des types numériques (entiers ou réels), le type booléen (avec seulement deux valeurs : le VRAI et le FAUX), et le type caractère. Un des rôles du compilateur est de vérifier la cohérence des types utilisés dans les opérations utilisant les variables.

L'identificateur d'une variable sert à référencer une variable, mais selon la manière dont on l'emploie, il peut aussi désigner la valeur contenue à l'adresse correspondante. La valeur étant interprétée en fonction du type de la variable, une variable doit toujours être **déclarée** avant d'être utilisée dans le programme pour faciliter le travail du compilateur.

La **déclaration** d'une variable dans le programme introduit son identificateur (son nom), et son type (celui de sa valeur). La déclaration du type des variables permet au compilateur d'effectuer les tests de cohérence, en particulier dans le calcul d'expression et/ou de fonctions qui les utilisent.

Ainsi par exemple en C, avant d'utiliser une variable entière appelée `degre`, on l'introduira par la déclaration :

```
int degre ;
```

qui définit `degre` comme variable de type `int` dont les valeurs possibles font partie d'un sous-ensemble des entiers.

Pour des raisons de performance, la taille de la zone mémoire allouée à une variable pour stocker sa valeur est fixe. Les langages de programmation distinguent donc plusieurs types numériques, catégorisés par la taille de la zone mémoire utilisée. Ainsi, la déclaration d'une variable indique finalement à la fois l'ensemble des valeurs qu'elle peut prendre, et la taille de la zone mémoire nécessaire pour stocker cette valeur.

Les valeurs manipulées sont exactes losrqu'on utilise des entiers, mais elles correspondent à des intervalles limités. Un domaine fréquent (entier signé codé sur 32 bits) est [-2^{31}, $+2^{31}-1$], soit un intervalle d'une taille de l'ordre de $2*10^9$. Si les valeurs obtenues par calcul dépassent la capacité de représentation, elles sont tronquées (on parle de **débordement**) et les résultats obtenus sont faux.

Pour les « réels », l'ensemble correspondant au type est plus vaste (de l'ordre de $1O^{37}$ pour des machines 32 bits) mais ne contient qu'un nombre fini de réels représentables (à peu près équirépartis logarithmiquement), les autres étant représentés par le réel le plus proche. L'erreur maximale ainsi introduite dans des calculs (en valeur relative) est en général 2^{-24}, c'est-à-dire de l'ordre de 10^{-7}.

Nous nous pencherons sur ces questions de représentations plus tard au second semestre, et présenterons simplement plus loin les types de base existants en C.

1.3.2 Adresse

La position de la zone dans laquelle est stockée la valeur d'une variable en mémoire centrale s'appelle **l'adresse** de la variable. Deux variables définies simultanément ont toujours des adresses différentes. Mais cette adresse étant déterminée dynamiquement au moment de l'exécution du programme, c'est une donnée qui reste généralement inconnue. (Certains langages de programmation ne font d'ailleurs jamais référence aux adresses).

Il arrive cependant qu'on veuille mentionner une variable pour la désigner en tant qu'emplacement mémoire, et non pas en tant que valeur (c'est le cas en C). Cette situation se produit en particulier lorsque l'on définit des modules de traitement prenant en argument des variables dont on souhaite modifier les valeurs par ce traitement. Dans ce cas, on ne peut indiquer 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.

1.4 Instructions et programme

Le texte d'un programme écrit dans un langage de programmation impératif est constitué d'une série d'ordres ou **instructions** donnés à exécuter à la machine. Ces instructions sont regroupées par séries (ou blocs) en un ou plusieurs modules (des fonctions) utilisant des variables. De manière générale, quand une instruction est terminée, on passe à l'instruction suivante dans le texte du module de traitement.

La puissance d'une instruction étant très faible, un programme est constitué de nombreuses instructions dont l'exécution est, par contre, très rapide (plusieurs centaines de millions d'instructions par seconde). Certaines instructions permettent heureusement de décrire des répétitions de séries d'instructions, ce qui fait que le texte d'un programme peut rester relativement court. Tout l'art sera de trouver des traitements itératifs atteignant leur but via la répétition de traitements élémentaires simples.

On classe habituellement les instructions en trois catégories :

- Les instructions d'affectation de variables
- Les instructions d'entrée/sortie
- Les instructions de contrôle

1.4.1 L'affectation

L'instruction d'affectation est l'instruction de base des langages impératifs. Elle a pour effet d'affecter une valeur à une variable. Elle comporte deux parties : la désignation de la variable qui va recevoir une valeur, et une **expression** qui décrit comment calculer cette valeur (à partir de valeurs d'autres variables ou de résultats de calcul de fonctions). Par exemple,

```
rayon * sin (angle – 0.5)
```

est une expression dans laquelle '`0.5`' est une constante, '`rayon`' et '`angle`' deux variables, '`sin`' une fonction et '`*`' et '`–`' des opérateurs. Ainsi, les expressions acceptées par un langage de programmation sont assez analogues aux expressions algébriques que l'on rencontre habituellement en mathématiques, mais l'on verra qu'il existe aussi des expressions de type booléen ou caractère.

En C l'affectation est notée par le signe d'égalité '='. La syntaxe générale suivie par une instruction d'affectation est très simple : à gauche du symbole d'égalité figure l'identificateur de la variable à affecter ; à droite figure une expression (qui peut être complexe) et qui décrit le calcul à effectuer pour calculer la valeur à affecter à la variable. Un point-virgule termine cette expression pour indiquer qu'il s'agit d'une instruction. Le calcul de la valeur décrite par l'expression est effectué en premier, puis cette valeur est affectée à la variable, c'est-à-dire rangée (i.e. écrite) à l'adresse mémoire correspondante.

Ainsi, une variable entière `degre` introduite par sa déclaration, peut ensuite être initialisée par l'affectation d'une première valeur (ici `10`), de la manière suivante :

```
int degre ;

degre = 2*5 ;
```

1.4.2 Les instructions d'entrée/sortie

Les **instructions d'entrée/sortie** (ou de lecture/écriture) sont des opérations réalisées en C au moyen de fonctions prédéfinies (dans une **librairie** standard), et qui permettent au programme d'interagir avec l'extérieur. Sans ce type d'instructions, un programme ne serait guère exploitable par des utilisateurs. Les instructions d'entrée permettent de fournir des données de départ à un traitement, ou d'orienter le déroulement du programme en cours d'exécution. On peut lire des entrées en provenance de différents supports périphériques comme le clavier, ou un disque (grâce à des fichiers). Les instructions de sortie permettent inversement d'imprimer des résultats ou des données sur l'écran, une imprimante, ou de les sauvegarder sur disque dans un fichier.

Ainsi, l'instruction C

```
printf("Bonjour tout le monde !") ;
```

permettra d'écrire les caractères « bonjour tout le monde ! » sur la console de l'écran. On verra plus loin que la fonction `printf` permet aussi d'écrire les valeurs des variables du programme dans différents formats.

1.4.3 Les instructions de contrôle

Les **instructions de contrôle** sont des instructions destinées à rompre le déroulement en série d'instructions écrites les unes derrière les autres. Ces instructions agencent des blocs d'instructions et peuvent être imbriquées. Elles « contrôlent » littéralement le déroulement du programme, d'où leur nom. L'une des instructions de base des instructions de contrôle est **l'instruction conditionnelle**, qui soumet l'exécution d'un bloc d'instructions à un test préliminaire, permettant ainsi de sauter, sous certaines conditions, une portion de programme.

Ainsi, l'instruction C

```
if (angle > 0)
      resultat = mesure / angle ;
```

n'enchainera sur l'instruction d'affectation de la variable `resultat` que si la condition sur la valeur de angle (`angle >0`) est réalisée. Cela permet d'empêcher l'affectation de la variable `resultat` dans le cas malencontreux où `angle` serait nul. Ce cas, en effet,

provoquerait une erreur à l'exécution du calcul, du fait de la division par zéro de la valeur de la variable `mesure`.

D'autres instructions de contrôle permettent de mettre en oeuvre des mécanismes de répétition appelés **boucles.** Les boucles effectuent de manière répétée une même suite d'instructions pourvu qu'une certaine condition soit réalisée (comme par exemple, que la valeur d'une variable reste positive). La condition est alors testée avant chaque nouvelle exécution des instructions figurant dans la boucle jusqu'à ce qu'elle soit invalidée. Nous en verrons un exemple plus loin.

1.5 Système d'exploitation, fichiers

Pour pouvoir interagir avec un ordinateur, il faut que celui-ci soit en train d'exécuter un programme spécial, le **système d'exploitation**. Il existe diverses variétés de systèmes, par exemple UNIX, WINDOWS ou VMS. Le système d'exploitation est ce qui permet aux utilisateurs *d'exploiter* l'ordinateur. Il permet d'interpréter des **commandes** (éditer un texte, créer ou détruire un fichier). Une commande est l'exécution d'un programme et n'a donc rien à voir avec une instruction (un programme est constitué de nombreuses instructions). Pour être exécuté, le code d'un programme doit être chargé dans la mémoire centrale de l'ordinateur, et la tâche principale d'un système d'exploitation est précisément de charger en mémoire centrale le programme qui exécute la commande (ou le logiciel) en cours.

Les informations nécessaires au fonctionnement des programmes et du système sont stockées en mémoire permanente dans des collections de données appelées **fichiers**. Le système d'exploitation gère aussi les fichiers, car ce sont eux qui permettent d'accéder aux données et aux programmes. Un ensemble ordinateur + système d'exploitation s'appelle une **plateforme**.

Les contenus des fichiers sont de natures très diverses : modules de code de programme exécutables, modules de code partiel (ou code objet), représentations internes de nombres, de dessins, etc., et de nombreux fichiers sont essentiellement constitués de caractères texte (en particulier le texte **source** des programmes).

Les fichiers portent un nom qui peut être suivi (généralement après un point) d'un suffixe nommé l'**extension**. Le rôle d'une extension est de donner une indication sur la nature des données contenues dans le fichier (fichier exécutable, fichier texte, etc.). Le mécanisme d'organisation des fichiers permet de les regrouper en **répertoires** (*directories* en anglais) et sous-répertoires dans une hiérarchie de noms indiquant leur emplacement et permettant de les classer en fonction de besoins particuliers (divers modules pour un logiciel, des librairies de programmes, ou encore, des boîtes de fichiers de courrier électronique, etc.).

1.6 Compilation

Un **compilateur** est un logiciel qui traduit un texte écrit dans un langage de programmation donné dans un autre langage : le langage machine. Un ordinateur particulier ne comprend en effet qu'un langage codé qui lui est spécifique : son langage machine.

Le compilateur du langage C est un programme complexe qui effectue cette traduction en plusieurs étapes. Ces étapes sont exécutées par plusieurs programmes, mais l'utilisateur n'aura pas besoin de les lancer et utilisera une commande générale (ou bien il bénéficiera d'un environnement intégré de programmation, dans lequel il pourra éditer le texte de son programme et lancer la compilation ou l'exécution). Il est cependant utile d'en avoir connaissance pour une meilleure compréhension du langage.

Il existe en particulier une première phase de compilation qui est exécuté par un **préprocesseur**. Un préprocesseur est un programme de prétraitement qui passe avant un autre programme pour en modifier la source (ici le texte) d'entrée. Tous les compilateurs C sont munis d'un préprocesseur, et certaines directives du texte d'un programme s'adressent à lui.

Le compilateur permet en outre de produire des modules de codes en partie incomplets. Ce type de code s'appelle un **code objet** par opposition au **code exécutable** (entièrement binaire). Un module de code objet est un module dont le code n'est pas directement exécutable, car certains éléments (des adresses de variables) ne sont pas encore connus.

Pour produire un module exécutable, il faut lier ensemble différents modules objet comblant les lacunes des uns ou des autres au sein d'un même code. L'**éditeur de liens** est un programme, qui fait partie du logiciel de compilation, et qui permet de rendre exécutable la liaison de plusieurs modules objet, en vérifiant que toutes les variables sont correctement définies.

Ce mécanisme permet de mettre à la disposition des programmeurs des bibliothèques (ou **librairies**) de fonctions précompilées en code objet, et, plus généralement, d'écrire un programme à partir de plusieurs fichiers source qui pourront être mis au point et compilés séparément. En effet, à partir du texte d'un module de programme, on peut obtenir un module de code objet qui pourra être utilisé ailleurs (cas des fonctions définies dans une librairie de fonctions), ou symétriquement, faire référence à des variables ou des fonctions définies ailleurs dans un module de programme (cas des programmes utilisant une fonction de librairie, ou définissant eux-mêmes plusieurs modules de fonctions compilés séparémment).

La figure suivante montre comment s'effectue la compilation d'un programme utilisant les fonctions d'une librairie.

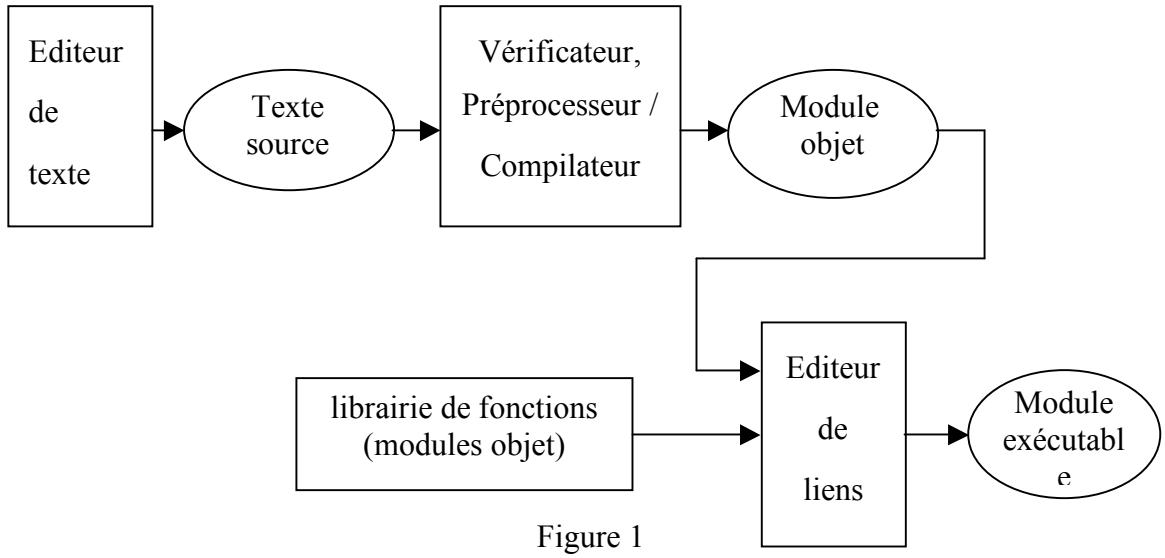


Figure 1
Compilation d'un programme utilisant une librairie de fonctions

Pour pouvoir utiliser les fonctions de la librairie, le module source du programme devra simplement en faire la déclaration en tête de fichier. Une procédure analogue s'applique aux variables utilisées par plusieurs modules dans le cas de la compilation en modules séparés. Les variables définies ailleurs devront être déclarées (avec la mention spéciale `extern` en C) en tête du fichier de module source avant d'être utilisées.

2 Eléments de base du langage C

2.1 Premiers programmes

La meilleure façon d'apprendre un nouveau langage est de lire des programmes commentés. Nous allons donc dans cette section introductive, parcourir les quelques notions introduites dans le chapitre précédent sur quelques exemples très simples.

2.1.1 « Bonjour tout le monde ! »

Un premier exemple servant toujours d'introduction dans l'apprentissage d'un langage de programmation, est le programme qui affiche à l'écran les mots « hello, world ». En voici une version française :

```
1   #include <stdio.h>
2
3   int main ( )
4   {
5      printf("Bonjour tout le monde !\n") ;
6      return 0 ;
7   }
```

Les numéros figurant à gauche sur chaque ligne nous permettront de les commenter une à une, mais ils ne font pas partie du texte du programme.

La première ligne figure dans la plupart des programmes C. Elle indique au compilateur qu'il faut inclure la déclaration des fonctions d'entrée/sortie de la librairie standard, car ce programme utilise l'une d'entre elles : la fonction `printf`.

Cette ligne commence par le symbole `#` car elle ne fait pas tout à fait partie du texte du programme : c'est en fait une directive adressée au préprocesseur. Elle commande l'inclusion automatique d'un fichier de librairie intitulé `stdio.h` qui contient la déclaration des fonctions de la librairie d'entrée/sortie. Tout le texte de ce fichier sera littéralement inclus ici, à la place de cette première ligne. L'inclusion est effectuée automatiquement, avant compilation, par le préprocesseur, modifiant ainsi le texte du programme qui sera effectivement compilé. Les déclarations contenues dans le fichier `stdio.h` sont nécessaires au compilateur pour vérifier que la fonction `printf` est correctement utilisée par le programme.

Ce fichier s'appelle `stdio.h` pour STandarD Input/Output, et son nom se termine par l'extension '`.h`' qui indique qu'il s'agit d'un fichier de déclarations. La seconde ligne est vide, mais permet de séparer le reste du programme de l'inclusion initiale des fichiers de déclarations, appelés pour cette raison, fichiers **d'en-tête**, parce qu'ils sont toujours inclus *en tête* du programme.

La troisième ligne introduit la définition de la fonction principale du programme. Cette définition s'étend jusqu'à l'accolade fermante (ligne7), et les instructions figurant entre les deux accolades (ligne 4 et ligne 7) sont les instructions qui seront exécutées par le programme.

Un programme consiste en un nombre arbitraire de variables et de fonctions (petits modules de traitement). L'une d'entre elles est primordiale et obligatoire pour l'obtention d'un code

exécutable : la fonction principale. C'est elle qui décrit ce que fait le programme - les autres n'étant définies que par commodité, pour être utilisées par elle. En C, on ne peut modifier le nom de la fonction principale qui s'intitule toujours `main`.

La définition d'une fonction comporte un bloc d'instructions figurant entre accolades, qui indiquent les opérations qu'elle va effectuer (plus éventuellement, on le verra plus loin, un ensemble de variables lui permettant de stocker des résultats intermédiaires). Le rôle d'une fonction mathématique est traditionnellement de fournir un résultat à partir des valeurs initiales de paramètres. Les fonctions d'un programme C sont conformes à ce schéma, puisqu'elles peuvent prendre en considération des paramètres de départ, et qu'elles retournent, après exécution de leurs instructions, une valeur. Mais elles sont susceptibles aussi d'avoir d'autres effets, comme ici, l'impression d'un message sur la console[2].

Dans le cas de la fonction `main`, des paramètres initiaux peuvent être fournis au lancement de la commande correspondant au programme compilé. Nous y reviendrons plus loin. Dans cet exemple, le programme lancé ne pourra pas prendre en compte de paramètres. En effet, les paramètres doivent, dans la définition d'une fonction, être introduits entre les deux parenthèses figurant après le nom de la fonction, et ici, ligne 3, le mot `main` est suivi simplement d'une parenthèse ouvrante et d'une fermante, sans paramètres interieurs.

De manière générale, la définition d'une fonction débute par le type de la valeur qu'elle retourne, et indique entre parenthèses la liste et le type de chacun des paramètres qu'elle prend en considération. Ici, la ligne 3 précise que la fonction `main` retourne un entier de type `int`, et qu'elle ne prend aucun paramètre. Mais nous ignorerons pour l'instant la valeur retournée par la fonction `main`. Dans le cas de toute autre fonction, la valeur retournée par une fonction est très importante, car elle sera utilisée ailleurs dans le programme (par exemple, dans une affectation de variable, ou pour fournir une valeur de départ à une autre fonction). Mais la fonction `main` est une fonction particulière, et la valeur finale qu'elle retourne ne peut être utilisée qu'au niveau de l'interprète de commande du système d'exploitation - niveau d'où on lance l'exécution du programme. Cette valeur doit nécessairement être un entier, mais la plupart du temps, elle sera ignorée du système.

Les lignes 5 et 6 situées entre les deux accolades constituent ce que l'on appelle généralement le **corps** de la fonction, et ici, parce qu'il s'agit de la fonction `main`, le **corps du programme**. Le corps d'une fonction contient le bloc d'instructions qui est exécutée par cette fonction. Les instructions qui y figurent finissent par un point-virgule et sont exécutées l'une après l'autre. C'est ce bloc d'instructions qui définit ce que fait la fonction, la ligne 3 ne déclarant que les types associés aux paramètres qui lui seront fournis en entrée, et le type de la valeur qu'elle retourne.

Ici, le corps du programme ne contient que deux instructions. La première,

```
printf("Bonjour tout le monde !\n") ;
```

est un appel à la fonction `printf` de la librairie d'entrée/sortie standard. Cette fonction prend ici un argument, `"Bonjour tout le monde !\n"`, qui est une **chaîne de caractères** (reconnaissable à cause des guillemets), et cette fonction a pour effet d'imprimer cette chaîne de caractères à la console. Les guillemets servent à délimiter les caractères de la

[2] Elles peuvent aussi orienter indirectement le reste de l'exécution du programme, car leurs paramètres peuvent ne pas être des valeurs, mais aussi des adresses de variables, qui pourront être modifiées.

chaîne. Le premier caractère imprimé à la console par cette fonction sera donc le caractère '`B`'.

Un caractère dans la séquence entre guillemets se représente lui-même, mais il arrive qu'on doive représenter des caractères « non imprimables » comme l'indication d'un passage à la ligne suivante sur la console, ou l'avancement jusqu'à une marque de tabulation.

Ces cas particuliers sont décrits par des séquences de caractères spéciaux. Ainsi, la séquence '`\n`' d'une chaîne de caractères indique un passage à la ligne suivante (*newline character*) et non pas la suite des deux caractères '`\`' et '`n`'. L'effet de la fonction précédente sera donc d'imprimer la suite 'Bonjour tout le monde !' à la console, puis de passer à la ligne suivante. Le dernier caractère visible sera donc le point d'exclamation et le prompteur ou l'impression suivante se trouveront sur une autre ligne.

Il est obligatoire d'utiliser la séquence '`\n`' pour inclure un caractère *newline* dans l'argument passé à la fonction `printf`. Si vous tentez d'écrire dans votre programme

```
printf("Bonjour tout le monde !
 ") ;
```

le compilateur signalera un message d'erreur et refusera de poursuivre son traitement. Par contre, on peut utiliser les trois instructions suivantes pour produire exactement le même effet :

```
printf("Bonjour ") ;
printf("tout le monde !") ;
printf("\n") ;
```

Notez que les deux caractères '`\n`' n'en représentent qu'un seul (le *newline*) dans la chaîne entre guillemets. Le caractère '`\`' est en effet interprété comme un caractère qui permet d'échapper au traitement standard (on parle de **caractère d'échappement**). Ainsi, le caractère '`\`' est en réalité ignoré, et a pour effet de modifier l'interprétation donné au caractère suivant. Ce mécanisme d'échappement permet de désigner des caractères invisibles. On en verra plus loin d'autres exemples, comme '`\t`' pour le caractère *tab* de tabulation,'`\b`' pour le caractère *backspace* qui permet de reculer, '`\"`' pour le guillemet (qui sans cela serait le signe que la chaîne à imprimer est terminée), et '`\\`' pour le *backslash* lui-même.

La dernière instruction est une instruction de **retour**,

```
return 0;
```

qui indique la fin de l'exécution de la fonction, et la valeur qu'elle retourne. Pour une fonction quelconque, ce résultat est rapporté dans le calcul où figurait l'appel de la fonction. Dans ce cas particulier, puisqu'il s'agit de la fonction principale, la terminaison de la fonction entraîne aussi celle du programme, et la valeur zéro est rapportée au système d'exploitation (là où a été lancé le programme).

Par convention, sous UNIX et sous WINDOWS, une valeur de retour nulle indique une terminaison correcte du programme. Cependant, dans le système VMS, la valeur de retour utilisée par convention pour signaler que tout s'est bien passé n'est pas `0`, mais `1`. On indiquera dans le troisième exemple comment écrire un programme général de ce point de vue.

2.1.2 Conversion de degrés Fahrenheit en degrés Celsius

Ce deuxième programme utilise la formule °Celsius = (5/9)(°Fahrenheit-32) pour afficher une table de correspondance entre degrés Fahrenheit et degrés Celsius :

```
0       -17
20      -6
40      4
60      15
80      26
100     37
120     48
140     60
```

Ce programme consiste encore en la définition d'une seule fonction (la fonction `main`), mais elle est un peu plus compliquée :

```
1   #include <stdio.h>
2
3/* imprime une table Fahrenheit-Celsius
4       pour fahr = 0, 20, ..., 140 */
5
6   int main ( )
7   {
8       int fahr, celsius;
9       int inf, max, ecart;
10
11      inf = 0 ;       /* temperature la plus basse */
12      max = 140 ;     /* temperature la plus haute */
13      ecart = 20 ;         /* ecart entre températures */
14
15      fahr = inf ;
16      while (fahr <= max)
17      {
18          celsius = 5 * (fahr-32) / 9 ;
19          printf("%d\t%d\n", fahr, celsius) ;
20          fahr = fahr + ecart ;
21      }
22      return 0 ;
23  }
```

Les lignes 3 et 4 contiennent un texte qui figure entre les signes '`/*`' et '`*/`'**.** Ce texte est un **commentaire**, sans incidence aucune sur l'exécution du programme, car il est complètement ignoré du compilateur. Les commentaires peuvent être introduits n'importe où (précisément, partout ou des caractères blancs peuvent être introduits) et peuvent figurer sur plusieurs lignes quand ils sont délimités par les signes '`/*`' et '`*/`'. La plupart des compilateurs acceptent aussi aujourd'hui des lignes de commentaires introduites par '`//`'. Dans ce cas, tout le reste de la ligne où apparaissent ces deux caractères est ignoré, et les commentaires du programme précédent auraient pu être introduits ainsi

```
1  #include <stdio.h>
2
3  // imprime une table Fahrenheit-Celsius
4  //  pour fahr = 0, 20, ..., 140
```

La ligne 6 qui introduit la définition de la fonction `main` est inchangée par rapport au programme précédent. Par contre, le bloc d'instructions qui définit le corps de la fonction utilise des variables. Ces variables doivent être déclarées avant d'être utilisées, et cette déclaration a lieu en tête de bloc, avant toute instruction exécutable. Les lignes

```
int fahr, celsius;
int inf, max, ecart;
```

déclarent ici les variables `fahr`, `celsius`, `inf`, `max` et `ecart`, comme variables entières de type `int`. Ces déclarations sont effectuées sur deux lignes distinctes par souci de clarification, afin de présenter ensemble les variables en rapport les unes avec les autres (du point de vue du sens du programme). Mais cet ordre est en réalité sans incidence sur le code.

L'exécution des instructions proprement dite commence ligne 11. Pour améliorer la lisibilité du programme, il est aussi fortement recommandé de sauter une ligne après la déclaration des variables, mais la ligne 10 est facultative et de manière générale, les lignes blanches sont ignorées. Le programme commence par initialiser les variables, c'est-à-dire leur affecter une première valeur avant d'exécuter tout calcul :

```
inf = 0 ;
max = 140 ;
ecart = 20 ;

fahr = inf ;
```

La dernière affectation attribue à la variable `fahr` la valeur de température la plus basse, c'est-à-dire celle de la variable `inf`, i.e. `0`, puisque la première instruction a affecté `0` à `inf` et que cette variable n'a pas été modifiée depuis.

Chaque ligne de la table est alors calculée avec la formule de la même façon, après avoir augmenté `fahr` de l'ecart de température souhaité pour la table, de sorte que le programme peut utiliser une boucle qui sera répétée pour chaque ligne affichée à la console. C'est le but ici de la boucle `while` :

```
while (fahr <= max)
{
          ...
}
```

Une boucle `while` opère de la façon suivante : la condition entre parenthèse est tout d'abord testée. Si elle est vraie (ici, si `fahr` est inférieur ou égal à `max`), le corps de la boucle, c'est-à-dire les trois instructions qui figurent lignes 18 à 20 entre accolades, sont exécutées. Ensuite, la condition est testée à nouveau, et, si elle est (encore) vraie, le corps de la boucle est exécutée à nouveau. Cette itération se poursuit jusqu'à ce que la condition devienne fausse (ici, jusqu'à ce que `fahr` soit strictement supérieur à `max`). L'exécution de la boucle est alors terminée, et le programme enchaîne sur l'instruction suivante, ici, `return 0` (ligne 22), et donc, dans ce cas, termine.

Les instructions qui constituent le corps de la boucle ont volontairement été disposées en retrait par rapport aux autres instructions, et notamment, par rapport au `while`. On peut ainsi voir en un clin d'œil les instructions qui seront répétées. S'il n'y avait eu qu'une seule instruction à répéter dans la boucle, cette instruction aurait figuré sans accolades, mais on l'aurait également disposée en retrait, comme dans

```
while (i < j)
```

```
        i  = i * 2 ;
```

Cette disposition s'appelle **l'indentation** car elle introduit un bord irrégulier (comme des dents) sur la partie gauche du texte. Elle n'est pas nécessaire pour la compilation du programme car les compilateurs ignorent complètement la disposition des instructions (une première phase de traitement consiste d'ailleurs à supprimer tous les blancs, tabulations, commentaires, et lignes inutiles). Mais elle est fondamentale pour la compréhension et la lecture du texte des programmes par un homme et non une machine.

Nous y attacherons une grande importance, et souhaitons que vos programmes soient toujours correctement indentés. Le texte d'un programme n'est en effet pas seulement destiné à la production d'un code par la machine. Ce texte doit aussi sans cesse être vérifié et modifié lors de la maintenance et de la mise au point du programme (par son auteur ou par d'autres), et il doit donc se présenter de la façon la plus compréhensible.

Il y a plusieurs manières de présenter l'indentation des instructions, et beaucoup de gens pensent que c'est une affaire de goût personnel. Mais l'expérience permet d'isoler les pratiques les plus intéressantes, et nous vous recommandons vivement d'adopter la méthode choisie dans ce cours. Nous vous recommandons en particulier de n'utiliser qu'une seule instruction par ligne, et d'utiliser des blancs autour des opérateurs pour clarifier les groupements. Une seule instruction par ligne est à prendre au sens large, c'est-à-dire qu'une même instruction pourra en réalité figurer sur plusieurs lignes ; par contre, aucune ligne ne doit contenir plusieurs instructions. La position des accolades est moins importante, bien que les gens se passionnent souvent pour cette question. Si vous n'adoptez pas le style utilisé ici, choisissez-en un autre parmi les styles les plus populaires, et utilisez-le de manière consistante et systématique.

En outre, quels que soient vos choix en la matière, nous vous recommandons impérativement d'éviter l'usage de caractères blancs pour réaliser l'indentation (on utilisera les caractères de tabulation). Les caractères de tabulation sont en effet beaucoup plus pratiques, parce que plus rapides à taper, plus sûrement exacts, mais aussi surtout parce qu'ils permettent de manière générale une réutilisation plus rapide des lignes introduites. Un groupe d'instructions peut toujours être réarrangé et réutilisé à un niveau d'indentation différent au sein d'un même programme, ou dans un autre programme. Les traitements de texte facilitent en effet beaucoup, grâce aux fonctions de Couper/Coller, la réutilisation de blocs d'instructions introduites ailleurs. Mais la différence de temps passé entre le réarrangement d'un texte produit avec des caractères blancs et celui produit avec des caractères de tabulations est tout à fait significative, et pénalise fortement les débutants qui l'ignorent.

Mais revenons au calcul effectué dans la boucle. Le correspondant en degrés Celsius de la valeur contenue dans `fahr` est calculé par l'instruction de la ligne 18

```
    celsius = 5 * (fahr – 32)  /  9;
```

La raison pour laquelle on n'a pas utilisé la formule `(5/9)*(fahr – 32)` est que dans ce cas, le résultat aurait toujours été nul. En effet, la division qui est effectuée entre `5` et `9` est une division entière, et le résultat de cette division est donc zéro, car la partie fractionnaire est ignorée. Pour obtenir quelque chose de plus significatif, on a reporté la division par `9` plus loin. Néanmoins, les résultats obtenus restent imprécis, car par exemple, le correspondant en degrés Celsius de `0°` Fahrenheit est plutôt `–17.8°` que `–17°`. Pour obtenir des résultats plus satisfaisants, il faut utiliser des variables à valeurs réelles, par exemple de type `float`. Nous ferons cette modification dans le troisième exemple.

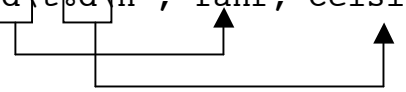
La ligne 19

```
printf("%d\t%d\n", fahr, celsius) ;
```

nous permet d'expliquer comment s'effectue l'impression avec `printf`.

On a vu dans le premier exemple que la fonction `printf` permettait d'imprimer une chaîne de caractères. Mais elle permet aussi d'introduire, au sein de cette chaîne imprimée, des formats pour des valeurs indiquées ensuite, en paramètres supplémentaires[3]. Pour cela, un caractère d'échappement fonctionnant de manière analogue au caractère '`\`' est utilisé dans la chaîne : le caractère '`%`'. L'apparition d'un '`%d`' signifie qu'on imprimera ici une valeur en format décimal (à cause du `d`), et chaque séquence précédée de '`%`' qui apparaît dans la chaîne est couplée avec l'expression suivante dans les arguments de `printf`. Ainsi, le premier '`%d`' correspond à une description du format d'impression pour la valeur du premier argument qui suit la chaîne de formet : ici la variable `fahr`, et le deuxième '`%d`' correspond à la description du format de l'argument suivant : ici la variable `celsius`.

```
printf("%d\t%d\n", fahr, celsius) ;
```



Cette fonction imprime donc ici d'abord la valeur (dans le système décimal) de `fahr`, puis un caractère de tabulation, puis la valeur décimale de `celsius`, puis saute à la ligne suivante (pour les impressions suivantes).

Il existe divers codes de la forme '`%x`' décrivant des formats de valeurs. En particulier '`%c`' permet d'imprimer une valeur de type caractère, et on aurait pu améliorer l'affichage de la table en rajoutant l'intruction suivante, en tête de programme (par exemple, ligne 10 ou 14) :

```
printf("%c\t%c\n", 'F', 'C') ;
```

On aurait alors obtenu l'affichage :

```
FC
0       -17
20      -6
40      4
...
```

Pour obtenir un meilleur alignement des nombres à droite, on peut préciser le nombre de chiffres qu'ils doivent occuper en largeur. Ainsi, l'ordre

```
printf("%3d  %6d\n", fahr, celsius) ;
```

n'utilise pas de tabulation mais permet d'avoir l'affichage suivant :

```
  0    -17
 20     -6
 40      4
 60     15
 80     26
100     37
```

---

[3] La plupart des fonctions prennent un nombre de paramètres fixé à l'avance, au moment de leur définition. La fonction `printf` est une exception : c'est une fonction **variadique**, c'est-à-dire, qui peut prendre un nombre quelconque (en réalité non nul en C) de paramètres effectifs, apparaissant au moment de l'appel de la fonction.

```
...
```

La dernière instruction de la boucle (ligne 20) est une affectation qui modifie la valeur de la variable `fahr` pour préparer le calcul et l'impression d'une autre valeur dans l'exécution suivante de la boucle :

```
fahr = fahr + ecart ;
```

Cette affectation utilise la variable `fahr` dans l'expression située à droite du signe d'affectation (`fahr + ecart`). Cette expression est calculée en premier, avec la valeur actuelle de `fahr`, puis rangée dans `fahr` comme nouvelle valeur. Ainsi, lors du premier passage dans la boucle, `fahr` vaut `0` et `ecart` vaut `20`. L'expression `fahr + ecart` vaut donc `20`, et c'est `20` qui est donc affectée à `fahr` par la première exécution de cette instruction. Au coup suivant, la condition `(fahr <= max)` est vraie car `max` vaut `140` et `fahr` vaut `20`. Les instructions de la boucle sont donc exécutées une seconde fois, avec les nouvelles valeurs. Cette deuxième exécution amènera la variable `fahr` à valoir `40` en fin de boucle. La condition `(fahr <= max)` sera donc encore vraie, et la boucle exécutée à nouveau. L'itération se poursuivra de la même façon jusqu'à ce que `fahr` prenne la valeur `140`. La boucle sera alors exécutée une dernière fois, mais cette fois, la dernière instruction ligne 20 affectera à `fahr` la valeur `160`, et la condition testée avant ré-exécution de la boucle sera fausse.

L'exécution de la boucle sera terminée (on dit que l'on « sort » de la boucle), et le programme enchaînera sur l'instruction suivante. Dans ce cas particulier il s'agit de

```
return 0 ;
```

et le programme termine.

Exercice

Pour suivre le déroulement de la boucle `while`, présenter une table donnant les valeurs des variables au cours d'une exécution du programme en indiquant sur chaque ligne le numéro de la ligne (dans le texte du programme) sur laquelle figure l'instruction qui a été exécutée avant l'obtention de ces valeurs.

2.1.3 Constantes symboliques

Les variables `inf`, `max` et `ecart` du programme précédent n'étaient guère très utilisées, et l'on aurait pu s'en passer en écrivant le corps de la fonction `main` de la manière suivante :

```
int main ()
{
   int fahr, celsius;

   fahr = 0;
   while (fahr <= 140)
   {
           celsius = 5 * (fahr-32) / 9 ;
           printf("%d\t%d\n", fahr, celsius) ;
           fahr = fahr + 20;
   }
   return 0 ;
}
```

Mais le programme aurait perdu en lisibilité, car les constantes entières `0`, `140` et `20` qui caractérisent les températures de la table n'apparaîtraient plus clairement. De manière générale, on évite de mettre des constantes et des nombres « magiques » dans les programmes. Les constantes qui sont utilisées ont généralement un sens, et l'on préfère utiliser un nom qui les symbolise pour que le texte du programme soit plus lisible, et plus facile à modifier ensuite.

Ainsi, une ligne commençant par `#define` permet de donner un nom à n'importe quelle suite de caractères. Cette directive suit la syntaxe suivante

```
#define   NOM   texte de remplacement
```

Ainsi

```
#define    MAX  140
```

définit le nom `MAX` comme désignant la séquence de caractères '`140`', c'est-à-dire en C, le nombre entier constant `140`.

C'est une directive adressée au préprocesseur qui lui demande de substituer le nom introduit par le texte de remplacement prévu, partout où ce nom figurera dans le programme. Le nom peut avoir la forme d'un identificateur de variable quelconque, mais on préfèrera par convention n'utiliser que des lettres majuscules et des caractères '_'. Le texte de remplacement n'est pas limité à des nombres, et l'on pourra ainsi définir aussi des chaînes de caractères constantes, ou des expressions plus complexes.

L'avantage de la définition de constantes ainsi nommées (appelées aussi **constantes symboliques** ou **constantes nommées**) est que si l'on veut modifier ces constantes, il suffit de faire la modification sur leur définition en tête de programme, et non pas à chaque endroit où elles apparaissent. On évite ainsi de parcourir tout le texte du programme, et on élimine le risque d'oublier des corrections.

Le programme précédent aurait pu être écrit de la manière suivante :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* programme de table Fahrenheit-Celsius */
5
6  #define INF   0     /* minimum de la table */
7  #define MAX   140   /* maximum de la table */
8  #define ECART  20   /* ecart entre temperatures */
9
10 int main ( )
11 {
12     int fahr;
13     int celsius;
14
15     fahr = INF;
16     while (fahr <= MAX)
17     {
18         celsius = 5 * (fahr-32) / 9 ;
19         printf("%3d %6d\n", fahr, celsius) ;
20         fahr = fahr + ECART;
21     }
```

```
22    return EXIT_SUCCESS ;
23 }
```

Dans cette nouvelle version, on a remplacé les variables entières `inf`, `max` et `ecart` par des constantes symboliques. On a en outre effectué l'inclusion du fichier d'en-tête de la librairie standard `sdtlib.h` pour utiliser la constante symbolique `EXIT_SUCCESS` qui s'y trouve prédéfinie[4]. On écrit ainsi un programme indépendant du système d'exploitation utilisé, car cette variable est définie correctement par la valeur signalant une terminaison correcte selon la plateforme (`0` pour une plateforme WINDOWS ou UNIX, et `1` pour une plateforme VMS).

Notez que les lignes en `#define` ne finissent pas par un point-virgule. En effet, il ne s'agit pas d'instructions, et ici, la constante symbolique `INF` est un nom pour la séquence constituée du seul caractère '`0`', et non pas pour la séquence des trois caractères ' `0 ;`'.

Enfin, dernière remarque : on aurait pu en réalité supprimer la variable `celsius`, car la fonction `printf` permet en réalité d'imprimer la valeur de n'importe quelle expression (et pas seulement celle d'une variable). Les deux lignes

```
18   celsius = 5 * (fahr-32) / 9 ;
19   printf("%3d %6d\n", fahr, celsius) ;
```

auraient donc pu être remplacées par

```
18   printf("%3d %6d\n", fahr, 5 *(fahr-32)/ 9);
```

et la variable `celsius` aurait ainsi pu être éliminée.

2.1.4 Lecture d'entrées

On peut modifier facilement les constantes symboliques d'un programme, mais celui-ci doit ensuite être recompilé si on veut que la modification soit prise en compte. Si le programme est destiné à être utilisé avec des valeurs différentes de température maximale ou d'écart, il vaut mieux utiliser des variables, et permettre à l'utilisateur d'en fournir les valeurs à l'exécution . La fonction `scanf` va nous permettre de réaliser cette amélioration :

```
1 #include <stdio.h>
2
3 /* programme de conversion Fahrenheit-Celsius */
4
5 #define  INF    0
6
7 int main ( )
8 {
9     int fahr ;
10    int max, ecart;
11
12    printf("Max et ecart des temperatures :");
13    scanf("%d", &max);
14    scanf("%d", &ecart);
15
16    fahr = INF;
```

[4] Ce fichier comporte aussi la définition d'une autre constante, `EXIT_FAILURE`, qui permet de terminer le programme en signalant qu'une erreur a été détectée.

```
17    while (fahr <= max)
18    {
19          printf("%3d %6d\n", fahr, 5 * (fahr-32) /
      9) ;
20          fahr = fahr + ecart;
21    }
22    return 0;
23 }
```

La fonction `scanf` ligne 13 permet de lire des caractères tapés sur la console dans le format précisé par le premier argument de cette fonction (ici la chaîne de caractères `"%d"`), et de ranger (i.e. affecter) ensuite la valeur lue dans une variable. Cette fonction n'est pas une fonction du langage, mais est définie dans la même librairie standard d'entrée/sortie que `printf`. Son fonctionnement est assez analogue : le premier argument de la fonction est une chaîne de caractères qui précise le format dans lequel la (ou les) valeur(s) doi(ven)t être lue(s). Ici, `"%d"` indique qu'il s'agit d'un entier (entré en notation décimale). Cette valeur est alors rangée à l'adresse qui suit dans l'appel de la fonction : ici, celle de la variable `max`.

On notera que l'on doit utiliser avec `scanf` l'opérateur d'adresse '`&`' devant le nom de la variable qu'on souhaite initialiser[5]. C'est assez naturel, car on souhaite ici désigner la variable comme zone mémoire (et non pas la valeur qui s'y trouve). Les indications de format pour lire des entrées avec `scanf` sont assez analogues aux indications de format de `printf`, et on peut lire plusieurs valeurs destinées à plusieurs variables dans le même appel à la fonction. Ainsi, on aurait pu demander la lecture des deux valeurs pour affecter `max` et `ecart` avec

```
scanf("%d%d",&max, &ecart) ;
```

ou la lecture d'une variable entière et d'un réel par exemple, avec

```
scanf("%d%f",&max, &celsius) ;
```

Si la chaîne de format contient des caractères particuliers, ils devront aussi figurer dans les caractères tapés à la console. Les blancs sont néanmoins toujours considérés comme un seul espace, et il n'est pas nécessaire de les préciser quand on veut saisir plusieurs nombres, car ils doivent alors nécessairement être introduits pour séparer les nombres entrés. La fonction `scanf` est pratique pour initialiser des variables mais parfois délicate à manipuler (cf. paragraphe 4.2.2).

Exercices

1) Modifier le programme pour imprimer la table en ordre inverse.
2) Ecrire un programme imprimant la table de conversion de degrés Celsius en degrés Fahrenheit.

2.2 L'alphabet et les mots du langage

Nous allons maintenant présenter de manière plus systématique les différents éléments constitutifs d'un texte de programme écrit avec le langage C. Pour définir un texte en langage C, on dispose des caractères suivants :

- Les lettres de l'alphabet usuel (en majuscules et minuscules) : A, B, ..., Z, a, b, ...z.

[5] Ce sera le cas pour toutes les lectures de nombres, mais pas pour les tableaux (ou chaînes) de caractères , comme nous le verrons plus tard.

- Les chiffres décimaux : 0, 1, ..., 9.
- Le caractère souligné (en anglais *underscore*) : _.
- Divers autres caractères :

  ! " # % & ‘ ( ) * + , - . / : ; < = > ? [ ] \ ^ { }
- Des caractères « blancs » : espace, tabulation horizontale, tabulation verticale, nouvelle ligne, nouvelle page.

Les mots avec lesquels on écrit des phrases dans un langage de programmation s'appellent des unités lexicales (ou lexèmes). Il y a 6 classes d'unités lexicales en C : les identificateurs, les mots-clés (ou mots réservés du langage), les constantes numériques ou de type caractère, les constantes de type chaîne de caractères, les opérateurs, et les délimiteurs ou séparateurs.

Les caractères blancs et les commentaires sont décrits globalement comme les « espaces blancs » et sont ignorés, sauf en ceci qu'ils séparent certains lexèmes et permettent leur identification. Ainsi, certains espaces blancs sont requis pour séparer des identificateurs qui se suivent, des mots-clés, ou des constantes. Sinon, les espaces blancs peuvent être utilisés librement, mais ils sont en réalité nécessaires pour rendre le programme plus lisible et plus compréhensible. Nous recommandons en particulier de mettre des blancs autour des opérateurs bien que cela ne soit pas requis par les règles du langage.

2.2.1 Identificateur

Un identificateur est un nom constitué d'une suite de lettres, de chiffres, ou de caractères soulignés '_', et qui commence par une lettre ou par un caractère souligné. Sa taille est limitée à un maximum de 31 caractères[6], et l'on ne peut pas choisir comme identificateur un mot-clé du langage, ceux-ci étant réservés à un usage bien particulier. On a vu que l'on utilisait des identificateurs pour nommer les variables, mais aussi des constantes (symboliques). On verra plus loin que des identificateurs sont également utilisés pour nommer des types définis par l'utilisateur, ou encore des fonctions.

Exemples

> `celsius`, `degre_celsius`, `degreCelsius`, et `_degre` sont des identificateurs valides. Par contre, `question?` , `auto` ou `#fg56` ne sont pas valides (`auto` est un mot réservé du langage).

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 et de fonctions par une minuscule. On évitera également les identificateurs commençant par '_', parce que beaucoup de fichiers d'en-tête de bibliothèques standard utilisent ce caractère pour préfixer des noms qui leur sont propres.

Certains caractères sont également à éviter, parce qu'ils peuvent être confondus avec d'autres, et une faute de frappe est alors difficile à repérer. Ainsi, on évitera d'utiliser le caractère zéro,

---

[6] Dans certains systèmes d'exploitation, les noms « externes » sont limités à moins de 31 caractères, et il se peut également que les minuscules soient alors traitées comme majuscules (c'est le cas du système VMS), mais ce n'est pas le cas sous UNIX.

à cause de la confusion possible avec la lettre O. De même, le chiffre 1 est très proche d'un L minuscule.

Il vaut mieux choisir des identificateurs explicites même s'ils sont un peu longs. Le caractère souligné permet de séparer des mots quand un identificateur en contient plusieurs, mais nous emploierons plutôt l'alternance majuscules/minuscules qui donne des identificateurs plus courts.

De manière générale, un identificateur doit toujours être le plus parlant possible. Il doit être lié au vocabulaire du problème analysé (i .e. celui apparu dans la phase d'analyse ou dans les documents de conception). C'est tout un art que de trouver de bons identificateurs, et c'est notamment à cette capacité « littéraire » qu'on reconnaîtra un bon programmeur.

2.2.2 Mots réservés

Voici la liste des mots réservés du langage C :

auto	default	float	register	struct	volatile
break	do	for	return	switch	while
case	double	goto	short	typedef	
char	else	if	signed	union	
const	enum	int	sizeof	unsigned	
continue	extern	long	static	void	

2.2.3 Constantes

Il y a différentes sortes de constantes. Pour les nombres, leur forme est grosso-modo une suite de chiffres et de lettres qui commence par un chiffre. Elle permet d'en déterminer le type précis. Nous les introduirons avec les types de base du langage dans la section suivante.

Les constantes de type caractère se composent pour la plupart du caractère référencé mis entre simples apostrophes. Ainsi,`'a'` désigne le caractère 'a' et `' '` le caractère blanc. Pour les caractères sans équivalent graphique, on utilise entre apostrophes une séquence de caractères commençant par le caractère d'échappement '`\`' suivi d'un ou plusieurs symboles particuliers. Ainsi, `'\t'` désigne le caractère de tabulation. On peut en outre, pour tous les caractères, faire suivre le caractère '`\`' du code du caractère dans la table des codes ASCII (voir plus loin). Ainsi, `'\048'` désigne le caractère '`0`' (ou chiffre zéro)**.**

Les constantes de type chaînes de caractères sont constituées d'une suite de caractères entourée de guillemets (ou *double quotes*). Dans cette suite, peuvent également figurer, pour les caractères spéciaux, des séquences commençant par le caractère d'échappement '`\`'. On en a déjà rencontré quelques exemples avec la fonction `printf`.

Exemples

`"une chaine", "\nchaine\tavec caracteres \" speciaux \" \n", "Un cas qui n\'est pas rare \?",` sont des chaînes de caractères constantes valides dont les deux dernières contiennent des caractères spéciaux.

La dernière chaîne contient des caractères spéciaux comme l'apostrophe et le point d'interrogation. La seconde comporte aussi des caractères spéciaux que nous avons déjà rencontrés : deux sauts à la ligne, un caractère de tabulation et des guillemets. Si on imprime

successivement la dernière et la seconde chaîne, on obtiendra l'affichage des deux lignes suivantes :

```
Un cas qui n'est pas rare ?
chaine     avec caracteres " speciaux "
```

La suite '`/*`' n'est pas considérée comme introduisant un commentaire lorsqu'elle se trouve dans une chaîne constante entre guillemets.

2.2.4 Opérateurs, délimiteurs et séparateurs

Les opérateurs, délimiteurs et séparateurs de C sont formés à partir des divers caractères qui ne sont pas utilisés par les identificateurs. La table de la Figure 2 indique toutes les séquences utilisées. On y trouve la plupart des opérateurs classiques comme `+` (addition) , `/` (division), etc. (ligne 1), des opérateurs de comparaisons (ligne 4) avec en particulier `!=` (différent de) et `==` (égal à).

+	-	*	/	%	&	|	^	>>	<<
=									
+=	-=	*=	/=	%=	&=	|=	^=	>>=	<<=
<	<=	==	!=	>=	>				
&&	||	!							
++	--	~							
,	;	?	:						
.	->								
(	)	[	]	{	}				
#	##								

Figure 2
Table des opérateurs, délimiteurs et séparateurs

Certains symboles combinent un opérateur et une affectation (lignes 2 et 3). D'autres, moins habituels, seront présentés plus loin. Enfin, # et ## sont des opérateurs qui s'adressent au préprocesseur.

2.3 Notation syntaxique

Dans la section précédente, nous avons fait la liste de toutes les unités lexicales du langage C. Pour écrire un texte de programme accepté par le compilateur, il faut utiliser ces « mots », mais également suivre les règles de **syntaxe** correspondant à la grammaire du langage.

Nous ne présenterons pas ces règles de manière exhaustive, car leur formulation systématique est un peu trop complexe (et inutile ici). Nous adopterons un point de vue intermédiaire, et pour introduire les formes correctes, nous noterons les catégories syntaxiques (par exemple, *type*, *expression*, *expression-composée*) en italique, et les caractères ou lexèmes litéraux (comme `>>`, `int`, etc.) avec la police `Courier` (caractères machine à écrire). Mais nous ne définirons pas toutes les catégories syntaxiques que nous mentionnerons, bien que cette construction soit relativement simple dans un langage de programmation.

3 Types, opérateurs et expressions

3.1 Types de base

On a vu que les variables à valeurs numériques ne pouvaient prendre qu'un sous-ensemble restreint de valeurs, du fait des limitations de la machine en matière de représentation. L'unité mémoire de base est le *byte*, qui correspond la plupart du temps à 8 bits (unité binaire), c'est-à-dire un octet. Ainsi, il n'y a que quelques types de base en C qui désignent :

`char`	un caractère (stocké sur 1 *byte*)
`int`	un entier (stocké sur la taille mémoire favorite d'entier de la machine, usuellement 2 ou 4 *bytes*)
`float`	un réel en précision simple (taille favorite de la machine)
`double`	un réel en double précision (taille la plus grande)

En outre, ces types de base peuvent être complétés d'un qualificatif qui en modifie le sens, comme `short` ou `long`, `signed` et `unsigned`.

3.1.1 Types entiers

Le type de base des entiers C est le type `int`, mais ce type n'est pas indépendant des machines car la taille de la zone mémoire allouée pour stocker une valeur de type `int` n'a pas été fixée par le langage à l'époque où il a été conçu . Des préfixes qualificatifs ont été prévus pour corriger ce défaut, et il existe en réalité plusieurs type d'entiers recouvrant différents domaines.

Les qualificatifs `short`, `long`, `signed` et `unsigned`, s'appliquent en effet au type `int`, pour modifier le domaine des valeurs représentées. Ainsi, on trouve des déclarations comme

```
short int sh ;
long int  compteur ;
```

Notons que dans ce cas, la mention du type `int` peut être supprimée. Notons aussi que `short` peut s'écrire `signed short int` ou `signed short` (même chose pour `long`).

Deux éléments sont à la base de la différenciation des types entiers du langage C : la taille de la zone mémoire utilisée pour stocker la valeur de la variable, et la nature des entiers représentés, i.e. naturels ou relatifs selon la terminologie mathématique, c'est-à-dire avec ou sans signe.

Le langage C ne spécifie pas la taille-mémoire des entiers de type `int` qui est de 2 ou 4 *bytes* selon les machines. Mais les qualificatifs `short` et `long` permettent de préciser des tailles mémoire de 2 ou 4 exactement. Par défaut, un `int`, un `short` ou un `long` représentent un entier relatif (signé), mais on peut utiliser le qualificatif `unsigned` pour modifier le domaine et ne représenter que des entiers naturels (sans signe). Les types `signed char` ou `unsigned char` sont également des types d'entiers de petite taille, car ils n'utilisent qu'un seul *byte*[7].

[7] Le type `char` sans qualificatif est utilisé pour décrire des valeurs de type caractère.

La table qui suit permet de trouver le domaine des valeurs selon le type, en combinant l'information sur la taille-mémoire et celle sur le signe (rien ou `signed` signifiant avec signe, et `unsigned` sans signe) :

Type	Taille mémoire
signed char, unsigned char	1 byte
(signed) short, unsigned short	2 bytes
(signed) int, unsigned int	2 ou 4 bytes
(signed) long, unsigned long	4 bytes ou plus

Domaines de valeurs : avec signe $[-2^{N-1}, 2^{N-1}-1]$, sans signe $[0, 2^{N}-1]$, où N est la taille en bits. Un *byte* correspond généralement à un octet, c'est-à-dire 8 bits.

Le fichier `limits.h` contient les constantes symboliques définissant les plus grandes et les plus petites valeurs disponibles sur la plateforme considérée pour chacun des types entiers[8]. Il faut prendre garde aux débordements avec les entiers signés, car le fait qu'ils soient signalés ou non dépend de la plateforme, mais généralement, ils ne sont pas signalés. (Pour les types non signés, cela peut être moins important car ils ont alors un comportement correct d'entiers modulo 2^{N}).

Les **constantes entières** peuvent être de type `int`, `unsigned int`, `long` ou `unsigned long`. Elles seront constituées d'une suite de chiffres en base octale si celle-ci commence par le chiffre zéro, hexadécimale si ce zéro est suivi d'un x (majuscule ou minuscule), et décimale sinon. Un code octal ne comporte ni `8`, ni `9`. Une suite hexadécimale peut contenir des lettres `a` ou `A` jusqu'à `f` ou `F` (qui codent les digits des valeurs 10 à 15). Les constantes entières peuvent également être suffixées (de lettres u ou L, indifféremment majuscules ou minuscules), pour préciser qu'il s'agit d'entiers non signés, ou qu'il s'agit de d'entiers de type `long`. S'il s'agit d'une représentation non suffixée, elle aura le type le plus court dans lequel sa valeur peut être représentée (`int` ou `long`).

Une **énumération** permet de définir une suite de constantes symboliques entières. C'est une alternative à l'utilisation de directives multiples en `#define`. En l'absence d'indication sur la valeur de ces constantes, la première vaut `0`, et les suivantes sont augmentées de 1 à chaque fois. Ainsi, par exemple,

```
enum { FALSE, TRUE }
```

défini `FALSE` comme valant `0`, et `TRUE` comme valant `1`. On peut aussi en préciser les valeurs avec un signe d'égalité (à ne pas confondre avec l'affectation : ici, on définit des constantes et non pas des variables) :

```
enum { VERT, BLEU, ROUGE = 5, ORANGE}
```

On aura alors : VERT = `0`, BLEU = `1`, ROUGE = `5` et ORANGE = `6`. Ces valeurs n'ont normalement aucune importance et peuvent même être égales.

---

[8] Signalons en particulier `INT_MIN`, `INT_MAX` pour les `int` signés, `UINT_MAX` pour les `int` non signés. On dispose aussi de `LONG_MIN`, `LONG_MAX` et `ULONG_MAX` pour les `long` et de `SHRT_MIN`, `SHRT_MAX` et `USHRT_MAX` pour les `short`.

La déclaration d'une énumération de constantes peut aussi constituer une définition de type, appelé **type énuméré,** si elle introduit un identificateur pour nommer l'ensemble des constantes définies. Ainsi la déclaration :

```
enum booleen {FALSE, TRUE} ;
```

permettra ensuite d'utiliser des variables de type `booleen`.

Exemple

`32767, 32768, 32768u, 0x7FFF,3L` sont des constantes entières valides, de types respectifs : int (qu'on suppose ici, sur 2 *bytes*), long, unsigned int, int et long.

Exercice

Donner le domaine des valeurs entières représentées à partir de la table, pour des `int`, `unsigned int`, `short`, `unsigned short` et `long`, `unsigned long` sur une machine 32 bits représentant les `int` sur 2 octets.

3.1.2 Types réels

Il y a trois types de réels prédéfinis : `float`, `double` et `long double`. La précision est d'au moins 6 chiffres décimaux significatifs pour le type `float`, et d'au moins dix chiffres pour les `double` et les `long double`. Le domaine est au minimum $[10^{-37}, 10^{37}]$ en valeur absolue. Pour ces types, le débordement est en général signalé.

On trouvera dans le fichier `float.h` des constantes symboliques définissant les plus petites et les plus grandes valeurs disponibles dans chacun des trois types[9].

Les constantes réelles peuvent être formées à partir de leur partie entière et de leur partie fractionnaire exprimées en décimal et séparées par un point symbolisant la virgule (notation anglaise). La partie entière ou la partie fractionnaire peuvent manquer. On peut aussi les donner sous forme d'une **mantisse** et d'un **exposant**. L'exposant est alors introduit après la mantisse par un E majuscule ou minuscule suivi d'un entier (éventuellement signé) :

`mantisse` `E` *`exposant`*

La valeur d'une constante introduite de cette manière est `mantisse * 10`$^{\texttt{exposant}}$. Cette notation (dite scientifique) est surtout utilisée pour des nombres très grands ou très petits.

Le type implicite des constantes réelles est `double`, mais il peut être modifié par un suffixe complémentaire optionnel. Le suffixe peut être F ou f (pour `float`), L ou l (pour `long double`).

Exemples

`1.239, .0778, 32L , 32.0, 345E-12` sont des constantes correspondant à des types réels.

Utilisons maintenant un type réel pour améliorer la précision de notre table de conversion Fahrenheit-celsius :

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

---

[9] On dispose en particulier de `DBL_MIN` et `DBL_MAX` pour les plus petits et les plus grands `double` positifs, ibid de `FLT_MIN` et `FLT_MAX` pour les `float` (positifs).

```
2  #include <stdlib.h>
3
4  /* programme de table Fahrenheit-Celsius */
5
6  #define INF   0     /* minimum de la table */
7  #define MAX   140   /* maximum de la table */
8  #define ECART  20   /* ecart entre temperatures */
9
10 int main ( )
11 {
12    int fahr;
13    float celsius;
14
15    fahr = INF;
16    while (fahr <= MAX)
17    {
18        celsius = (5.0/ 9.0) * (fahr-32) ;
19        printf("%3d %6.1f\n", fahr, celsius) ;
20        fahr = fahr + ECART;
21    }
22    return EXIT_SUCCESS ;
23 }
```

Dans cette nouvelle version, on a modifié le type de la variable `celsius` pour avoir un réel et donc une meilleure précision. Dans le calcul de conversion ligne 18, l'expression

```
(5.0/ 9.0) * (fahr-32)
```

contient la variable `fahr`, de type `int`. Mais la valeur de `(fahr – 32)`, de type `int`, sera convertie automatiquement en `float` pour exécuter la multiplication par `(5.0/9.0)`. Les conversions de types sont en effet (souvent) réalisées automatiquement, mais on verra plus loin qu'il est parfois nécessaire de les forcer et de les expliciter. Ainsi, on aurait pu écrire l'instruction précédente différemment, comme ceci :

```
(5.0/ 9.0) * ((float)fahr-32.0)
```

La mention du type `float` entre parenthèses devant la variable entière `fahr` en force alors la conversion avant tout calcul, et on utilise ensuite la soustraction sur des réels.

Ligne 19, l'expression

```
printf("%3d %6.1f\n", fahr, celsius) ;
```

illustre un autre usage de la fonction `printf`: la première séquence de format '`%3d`' de la chaîne d'argument indique que la variable `fahr` doit être imprimée en notation décimale sur au moins 3 caractères, et la seconde séquence de format '`%6.1f`' indique que la variable `celsius` doit être imprimée sur au moins 6 caractères avec 1 caractère après la virgule. Le résultat des ordres d'impression du programme ressemble alors à ceci :

```
 0   -17.8
20    -6.7
40     4.4
...
```

La largeur et la précision du nombre de chiffres après la virgule peuvent être omises dans une spécification de format. Ainsi '`%6f`' indique que le nombre sera imprimé sur au moins 6

caractères ; '`%.2f`' qu'il comportera 2 chiffres après la virgule, et '`%f`' simplement qu'il s'agit d'un nombre de type `float`. Signalons aussi que `printf` comprend les formats '`%o`' pour un entier en base octale, '`%x`' pour une base hexadecimale, '`%c`' pour un caractère et '`%s`' pour une chaîne de caractères. (Pour échapper à cette interprétation du caractère '`%`', et imprimer un simple '`%`' (par exemple suivi d'un simple '`d`' !!), il faudra utiliser le code '`%%`').

3.1.3 Type caractère

Le type `char` est un type permettant de manipuler des variables dont les valeurs sont des caractères.

On a vu que les constantes de type caractère étaient constituées d'une suite de symboles entre apostrophes. Pour la plupart des caractères imprimables, cette suite est réduite au caractère considéré, mais il existe un certain nombre de séquences spéciales pour les caractères non imprimables ou ambigus, formées à partir du caractère d'échappement '`\`'. Ainsi, `'\n'` désigne le caractère de retour à la ligne (*newline*), et `'\t'` le caractère de tabulation horizontale. De même, `'\\'`, `'\''`, `'\"'`, `'\?'` sont des constantes caractères qui désignent respectivement les caractères '`\`' (*backslash*) , l'apostrophe '`''` (*single quote*), le guillemet '`"`' (*double quote)*, et le point d'interrogation.

On peut également écrire une constante caractère en utilisant directement le code octal ou hexadécimal du caractère dans le code ASCII (pour *American Standard Code for Information Interchange*). Notons que les caractères ASCII sont restreints à un ensemble de 128 caractères car ils sont codés sur 7 bits[10]. Les 32 premiers codes (codes hexadécimaux de 00 à 1F) représentent des caractères de contrôle qui ne sont pas affichables (ils servent de signaux dans la gestion des entrées/sorties). Parmi eux, le caractère NUL, qui a pour code zéro, servira à indiquer la fin d'une chaîne de caractères dans les échanges avec les mécanismes d'entrée/sortie. Les caractères de contrôle sont ensuite suivis de quelques caractères de ponctuation. Les caractères numériques sont rangés alors par ordre croissant (codes 48 à 57). S'ensuivent ensuite d'autres caractères imprimables, dont les lettres en ordre alphabétique (codes 65 à 90 pour les majuscules, 97 à 122 pour les minuscules).

Une séquence de type `'\`*ooo*`'` permet de désigner un caractère en indiquant son code ASCII en octal sur 3 digits maximum *ooo* (et une séquence `'\x`*hh*`'` permet de l'indiquer par son code en hexadécimal). Le caractère nul est donc `'\0'` (ou `'\000'`), et le *chiffre* zéro est `'\048'`. Signalons au passage un caractère amusant : le caractère de code octal `007`. Ce caractère est le caractère *beep*, dont l'affichage produit un avertissement sonore. (Ce caractère possède aussi un code symbolique : `'\a'`).

[10] Ce code a été adopté historiquement de façon quasi universelle en code interne sur les machines, et beaucoup de codes récents en sont des extensions. Il existe cependant un autre code (EBCDIC) qui n'a pas les mêmes propriétés. Le code ASCII a été conçu à l'origine pour les besoins de l'informatique en langue anglaise, et il ne permet pas de coder les caractères accentués du français, ni les caractères spéciaux d'autres langues comme l'arabe ou le chinois. D'autres codage des caractères ont été proposés depuis dans certains langages. En particulier, le langage java utilise des caractères unicodes, codés sur 16 bits, permettant de manipuler de nombreux caractères dans différentes langues.

3.1.4 Autres types

Il existe des types dérivés des types de base que nous présenterons plus loin (tableaux, structures, unions, pointeurs et fonctions) ; mais le langage ne dispose pas d'autres types simples.

a. Type booléen

Il n'existe pas en C de type booléen permettant de manipuler des valeurs spéciales (VRAI, FAUX). On utilise à la place un type entier, avec comme convention qu'une valeur nulle représente le FAUX, et qu'une valeur non nulle représente le VRAI. On verra plus loin que le langage fournit néanmoins des opérateurs booléens (qui retournent la valeur `0` pour représenter le FAUX, et la valeur `1` pour représenter le vrai). Il existe aussi des opérateurs de comparaisons (`==` pour l'égalité et `!=` pour la non égalité) permettant de comparer deux valeurs de n'importe quel type de base (numérique ou caractère).

Ainsi, le dialogue avec l'utilisateur pour entrer des données dans la dernière version du programme de conversion Fahrenheit-Celsius pourra être améliorer de la façon suivante :

```
5   int main ( )
6   {
7      int fahr ;
8      int max, ecart;
9      char reponse;
10
11     /* debut du dialogue avec l'utilisateur */
12     printf("Voulez-vous modifier max (=120)\n");
13     printf("et ecart (=20) \?\n");
14     printf("Repondez O pour oui et N pour non :");
15     scanf("%c", &reponse);
16
17     /* initialisations de max et ecart */
18     if (reponse == 'N')
19     {
20        max = 140 ;
21        ecart = 20 ;
22     }
23     else
24     {  /* reponse oui (ou autre ...) */
25        printf("Entrez la valeur maximum et ");
26        printf("l\'ecart des temperatures :");
27        scanf("%d%d", &max, &ecart);
28     }
29     ...
```

b. Type chaine de caractères

Bien que le langage C reconnaisse des constantes de type chaînes de caractères, le langage ne propose pas le type de base chaîne de caractères. Les chaînes de caractères peuvent cependant être traitées en C par des tableaux de caractères. Le type tableau est un type dérivé des types de base, permettant de manipuler une variable pouvant contenir plusieurs valeurs (ordonnées)

d'un même type. Il existe un type tableau de caractères couramment utilisé : le type `String`, qui est défini par-dessus le langage dans une librairie[11], mais nous ne l'introduirons pas spécifiquement.

c. Type `void`

Il existe un type particulier, spécifique au C, dont le domaine de valeurs est vide : le type `void`. Ce type sert à déclarer le type de la valeur de retour d'une fonction lorsque celle-ci ne retourne en réalité aucune valeur. (On verra plus tard que ce type est également utilisé pour déclarer un type de pointeurs universels).

3.2 Opérateurs et expressions

Le rôle des expressions est de spécifier des calculs ou des transformations à appliquer aux données pour rendre un résultat. Ce résultat est d'un certain type, considéré comme le type de l'expression. La plupart du temps, les expressions servent à affecter de nouvelles valeurs à des variables. Le type de l'expression est alors comparé au type de la variable par le compilateur qui peut générer un message d'erreur (en cas de mauvais appariement), ou effectuer les conversions implicites de types permettant l'affectation.

Une expression se compose d'**opérateurs**, d'**opérandes** et de **délimiteurs**. Les opérateurs sont des lexèmes associés à des opérations (par exemple + pour l'addition et % pour le reste modulo d'une division), les opérandes sont des éléments sur lesquels portent les opérations (variable, constante, appel de fonction, résultat intermédiaire). Les délimiteurs vont par paire (ouvrant et fermant). Ainsi,

```
(4 – 2) * 5
```

est une expression qui comporte deux opérateurs (`-` et `*`) et deux délimiteurs (les parenthèses). Les opérandes de la soustraction sont les constantes `4` et `2`, et ceux de la multiplication sont l'expression `(4-2)` et la constante `5`. Le résultat du calcul de cette expression est `10`, mais celui de l'expression correspondante sans parenthèses est `–6` :

```
4 – 2 * 5
```

En effet, la multiplication étant prioritaire par rapport à la soustraction, `2 * 5` est effectuée en premier.

Les ordres de priorité entre opérateurs déterminent l'ordre d'exécution des calculs (en l'absence de parenthèses). Pour des opérateurs identiques ou de même niveau de priorité, c'est l'associativité (à gauche ou à droite), qui détermine si les opérations seront effectuées de droite à gauche ou de gauche à droite. Ainsi, le résultat du calcul de l'expression :

```
3 – 2 + 5
```

est 6, parce que ces opérations sont **associatives à gauche**, c'est-à-dire qu'elles sont effectuées de gauche à droite : d'abord la soustraction `3 – 2`, qui donne `1`, puis l'addition qui donne `6`. Mais si ces opérations avaient été associatives à droite, on aurait obtenu un tout autre résultat : l'addition `2 + 5` d'abord donne `7`, qui retranché à `3`, donne `-4`. *Nous recommandons donc vivement, en cas de doute, d'utiliser des parenthèses*. Ce sont des

[11] Il faut inclure le fichier d'en-tête `string.h` pour utiliser des fonctions de cette librairie.

délimiteurs optionnels, mais ils forcent l'ordonnancement des calculs et clarifient l'expression.

Remarque importante

La plupart des opérateurs sont associatifs à gauche. Mais cette propriété n'indique que l'ordre dans lequel les opérations sont effectuées, et non pas dans quel ordre leurs opérandes seront calculés. *L'ordre d'évaluation des opérandes n'est pas précisé par le langage*, et cela peut conduire à des résultats différents selon les compilateurs. (De même, les arguments d'une fonction ne sont pas toujours évalués de gauche à droite !). Certains opérateurs évaluent néanmoins toujours leur opérande gauche d'abord : il s'agit des opérateurs (ET et OU) logiques, de l'opérateur conditionnel '`?:`' et de l'opérateur de séquencement '`,`' (voir plus loin).

3.2.1 Opérateurs numériques et logiques

La table de la Figure 3 présente les différents opérateurs numériques et logiques par ordre de priorité décroissante. La négation logique située en haut est l'opération la plus prioritaire. Tous les opérateurs binaires situés en dessous sont associatifs de gauche à droite.

Opérateurs	Significations
!	négation (NON) logique
* / %	multiplication, divisions et opérateur modulo
+ –	addition, soustraction
< <= > >=	comparaisons
== !=	comparateurs d'identité
&&	ET logique
||	OU logique

Figure 3
Opérateurs numériques et logiques

Les opérateurs arithmétiques s'appliquant aux entiers sont `+`, –, `*`, `/`, et l'opérateur modulo `%`. La division `/` entre entiers supprime la partie fractionnaire, et l'expression

```
x % y
```

retourne le reste de la division de `x` par `y` (elle ne vaut donc zéro que si `y` divise `x` exactement). L'opérateur modulo `%` n'est défini que pour des entiers et ne peut être appliqué à des `float` ou des `double`. Par contre l'addition, la soustraction, (et l'opérateur unaire – donnant l'opposé d'un nombre), ainsi que la multiplication et la division, peuvent être effectuées avec des entiers ou des réels.

Exemple

Une année bissextile est une année qui est divisible par `4` et pas par `100`, mais les années divisibles par `400` sont considérées comme bissextiles.

```
if ((annee % 4 == 0 && annee % 100 != 0)|| annee % 400 ==
0)
      printf("%d est une annee bissextile\n", annee);
else
```

```
        printf("%d n\'est pas une annee bissextile\n",
annee);
```

Les opérateurs `&&` et `||` correspondent aux opérateurs booléens de la logique classique, le Λ (ET), et le V (OU inclusif). Ces opérateurs agissent sur des opérandes booléens et sont définis par la table de la Figure 4, qui donne respectivement les valeurs de A `&&` B et de A `||` B en fonction des valeurs de A et de B (qui peuvent être VRAI ou FAUX). L'opérateur ' `!` ' est l'opérateur de négation. Il transforme le FAUX en VRAI et le VRAI en FAUX. Ces opérateurs permettent de créer des expressions booléennes qui sont utilisées dans des instructions conditionnelles, comme dans l'exemple qui précède.

On a déjà mentionné que le type booléen n'existant pas en C, le résultat de ces opérateurs est en réalité un `int`. Ces opérateurs retournent néanmoins `0` pour indiquer le FAUX et `1` pour indiquer le VRAI. Mais leurs opérandes peuvent prendre des valeurs entières quelconques, *toute valeur non nulle étant considérée comme* VRAI.

A	B	A `&&` B	A `||` B	`!`A
FAUX	FAUX	FAUX	FAUX	VRAI
FAUX	VRAI	FAUX	VRAI	
VRAI	FAUX	FAUX	VRAI	FAUX
VRAI	VRAI	VRAI	VRAI	

Figure 4
Table de vérité des opérateurs booléens

Remarque

Signalons que ces opérateurs ne calculent leur opérande droit que si sa valeur peut modifier le résultat, une fois calculée la valeur de l'opérande gauche. Ainsi, si A est évalué à FAUX, B ne sera pas évalué dans le calcul de l'expression A && B, car le résultat sera toujours FAUX, indépendamment de la valeur de B. De même, si A est évalué à VRAI, B ne sera pas évalué dans le calcul de l'expression A `||` B, qui rapporte VRAI directement. Bien exploité, ce comportement peut être utile, mais il peut aussi entrainer des erreurs, si l'expression non évaluée modifie une variable.

Les opérateurs `==` et `!=` sont des opérateurs de comparaisons qui retournent également une valeur « booléenne », valant `0` ou `1`. Ces opérateurs de comparaisons, ainsi que leurs homologues `<`, `>`, `<=` et `>=`, existent sur les entiers comme sur les réels, mais on se méfiera des résultats obtenus avec des réels, par suite de l'imprécision sur leur mode de représentation. Ainsi, sur beaucoup de machines, `1./10*10` n'est pas égal à `1`, mais à `0.999999`... (parce que `0.1` ne peut être représenté exactement).

Les caractères étant codés par des entiers, les opérateurs arithmétiques et les opérateurs de comparaison sont également définis sur le domaine des caractères. Pour ces opérations, la valeur de la variable est traitée par la machine comme celle de l'entier qui l'encode. Le fait que les caractères chiffres ou lettres soient codés par des entiers consécutifs (c'est le cas dans le code ASCII), permet alors d'effectuer des tests ou des calculs appropriés en utilisant ces opérateurs. Ainsi, si `c` est un caractère chiffre (c'est-à-dire, s'il vaut `'0'`, `'1'`, ... ou `'9'`), le résultat de la soustraction

```
c - '0'
```

sera l'entier correspondant au chiffre que le caractère désigne. Cette propriété sera exploitée par les fonctions de conversions de caractères entrés au clavier en nombre. De même, le test

```
if (c >= '0' && c <= '9')
```

permet de savoir si le caractère `c` est un chiffre, et

```
if (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z')
```

permet de savoir si le caractère `c` est une lettre.

Exercice

Ecrire un programme qui lit au clavier deux caractères `d` et `u` et imprime à la console le nombre décimal qui s'écrit '`du`', c'est-à-dire dont `d` est le chiffre des dizaines et `u` celui des unités.

3.2.2 Conversions de types

a. Conversion implicite

Les opérandes d'un opérateur numérique sont généralement du même type, et ce type est le type du résultat (sauf pour les opérateurs de comparaisons). Par exemple :

```
9 / 4
```

donnera `2` comme résultat, puisque `9` et `4` sont des `int`. On obtient en effet alors un résultat de type `int`. De même :

```
9.0 / 4.0
```

donnera `2.25`, de type `double`. Mais, si l'on écrit

```
9 / 4.0   (ou  9.0 / 4 )
```

`9` de type `int` sera converti en `9.0` de type `double` et le résultat sera `2.25`, de type `double`. De manière générale, les règles de C sont de convertir l'opérande du type le plus « étroit » en celui du type le plus « large » et de retourner une valeur du type le plus large. Les inclusions de types, en allant du plus étroit vers le plus large, sont les suivantes :

char short int long float double long double

C autorise aussi des mélanges de types `signed` ou `unsigned`, mais les résultats peuvent être parfois étranges (et ils sont dépendants de la plateforme). Nous recommandons donc de les éviter.

b. Conversion forcée (casting)

L'opérateur de conversion de type (*cast* en anglais) est constitué d'un couple de parenthèses ouvrante et fermante entourant le (nouveau) type dans lequel s'effectue la conversion. On le place devant l'expression dont on souhaite modifier le type. Il faut le manipuler avec précaution, car il peut dans certains cas causer une perte d'information. On l'utilise de manière standard pour convertir des réels en entiers (mais il ne faut pas que ces réels soient trop grands !).

Exemple :

```
int x;
float y = 8.5;
```

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

L'opérateur de *cast* sur `y` qui convertit la valeur de `y` en `int` permet d'appliquer l'opérateur `%` (défini sur les entiers) à une variable réelle. 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 fournit le premier opérande pour l'opérateur modulo.

On utilise souvent une conversion forcée pour récupérer une valeur du bon type, par exemple une constante (définie comme `int`) :

```
unsigned short int x ;
x = (unsigned short int) VERT ;
```

Un autre cas fréquent est de forcer la conversion au type `void` pour ignorer le résultat d'une fonction (le compilateur émet sinon un message pour signaler l'erreur), lorsqu'on souhaite appeller une fonction, non pas pour en récupérer la valeur, mais pour les effets qu'elle provoque par ailleurs :

```
(void) getchar() ;
```

Ici, la fonction `getchar`, qui rapporte normalement un caractère entré au clavier, est utilisée pour le fait d'avancer d'un caractère sur le flot des entrées, mais ce caractère est inutilisé dans le programme (on le « saute »), et la valeur de retour de la fonction peut être ignorée.

3.2.3 Opérateurs de modification de variables : les affectations

L'opération de modification par excellence est l'affectation, mais il en existe plusieurs variantes. Les affectations sont regroupées sous le terme général *d'opérateurs* de modification, car elles calculent une valeur et peuvent à ce titre figurer dans n'importe quelle expression. Mais il ne faut pas perdre de vue que ces opérateurs ont un effet supplémentaire : celui de modifier la valeur de la variable sur laquelle ils portent, et la plupart du temps, les expressions d'affectations seront destinées à devenir des instructions. On peut en effet, comme nous allons le voir plus loin, transformer de telles expressions en **instructions** en leur ajoutant simplement un point-virgule.

a. Affectations de variables

Comme nous l'avons déjà dit, la syntaxe générale d'une expression d'affectation de variable est très simple : elle utilise le signe égal '=' en plaçant à gauche un unique identificateur de variable, et à droite, une expression quelconque :

identificateur `=` *expression*

L'effet d'une affectation de variable est de placer, à l'adresse de la variable considérée, le résultat du calcul de l'expression. Par exemple la valeur de la variable `nb` de type `int` peut être modifiée par les deux instructions d'affectations successives de la manière suivante

```
nb = 3 ;
            /* La valeur de nb est maintenant 3 */
nb = nb + 2 ;
            /* La valeur de nb est maintenant 5 */
```

Soulignons sur la dernière affectation un point important. *Dans la partie à gauche* du symbole d'égalité, l'identificateur `nb` désigne la variable *en tant qu'emplacement mémoire*, alors que dans la partie droite, il figure dans une expression et désigne simplement sa valeur (ici `3`).

Cette syntaxe particulière permet au langage C d'accepter des **affectations multiples**. Dans ce cas, on enchaîne plusieurs symboles d'égalité entre identificateurs de variables, et seule la partie la plus à droite contient une expression. L'effet d'une affectation multiple est de donner la valeur du calcul de l'expression située à droite à toutes les variables mentionnées à gauche. Ainsi

```
x = y = nb + 1
```

a pour effet d'affecter à la variable `x` *et* à la variable `y` la *valeur* de la variable `nb` plus 1.

Remarque

Le langage C accepte aussi qu'une valeur soit affectée à une variable au moment de sa déclaration. On dit alors que la variable est **initialisée** au moment de sa déclaration. Par exemple, la déclaration suivante introduit un entier `x` de type `int`, initialisé à `10` :

```
int x=10 ;
```

Une autre originalité du langage C est la présence d'opérateurs d'**affectation composée**. Ce sont en fait des abréviations pour des instructions de la forme

`ident = ident` *`op`* *expression*

où `ident` désigne un identificateur de variable quelconque, mais se trouve placé à la fois à gauche et à droite du signe d'affectation ; *`op`* désigne un opérateur et *expression* une expression quelconque. De telles affectations, où la valeur de la variable affectée apparait à droite en tant qu'opérande gauche d'un opérateur, sont abrégées en

`ident` *`op`*`=` *expression*

où le symbole de l'opérateur se trouve accolé au signe d'égalité à sa gauche. Ainsi `x += 3` est équivalent à `x = x + 3`, de même, `x /= 2` signifie `x = x / 2`, etc.

b. Incrémentation et décrémentation

Le langage C fournit deux opérateurs inhabituels (que l'on retrouve en java et en C++) pour incrémenter ou décrémenter des variables. L'opérateur `++` ajoute 1 à son opérande, tandis que l'opérateur `–` lui retranche 1. Ces opérateurs ne s'appliquent qu'à des variables, et une expression comme `++(i+j)` est illégale.

Il y a deux façons distinctes d'utiliser ces opérateurs : comme opérateur *préfixe* (comme dans `++n`) ou comme opérateur *suffixe* (comme dans `n++`). Dans les deux cas, l'effet est d'incrémenter la valeur de `n`, mais dans le cas préfixe `++n`, cette opération est effectuée *avant* que cette valeur ne soit utilisée par le calcul de l'expression où elle figure, tandis que dans le cas suffixe `n++`, elle est effectuée *après*. Ainsi, si la variable entière `n` vaut `5`, l'effet de

```
x = n++
```

sera différent de celui de

```
x = ++n
```

Dans les deux cas, la valeur de `n` passera de `5` à `6`, mais le résultat final sera différent pour `x`. La première affectation est en fait équivalente à la séquence

```
x = n ;
n = n + 1
```

alors que la seconde est équivalente à

```
n = n + 1 ;
x = n
```

Attention

Il faut éviter de faire figurer deux fois dans une même expression une variable modifiée par incrémentation, car la valeur de cette variable n'est spécifiée par le langage qu'après calcul de l'expression qui la contient, mais il y a ambiguïté sur cette notion lorsqu'il y a plusieurs sous-expressions. (En outre, on ignore la plupart du temps l'ordre de calcul des opérandes). Ainsi, dans le calcul de

```
i++  -  (n * i)
```

on sait que l'incrémentation de `i` aura lieu après calcul de la soustraction, donc, on connait la valeur de ce premier opérande, mais on ignore si la modification sur `i` sera effectuée avant son utilisation par la multiplication ou plus tard, en fin de calcul, ce qui fait qu'on ignore la valeur du second opérande.

3.3 Types dérivés des types de base

A côté des types de base, il y a conceptuellement une infinité de types dérivés construits à partir des types fondamentaux suivants, chacun opérant d'une certaine manière :

Tableaux	d'objets d'un type donné
Fonctions	retournant un objet d'un type donné
Pointeurs	sur des objets d'un type donné
Structures	contenant une liste d'objets de types divers
Unions	pouvant être un objet d'un type quelconque parmi plusieurs

Nous introduirons plus loin ces différents types, mais nous allons donner d'ores et déjà quelques exemples, destinés à fournir suffisamment de matériel pour pouvoir commencer à écrire des programmes qui ne soient pas trop triviaux.

3.3.1 Types définis par l'utilisateur

La partie déclarative d'un programme C peut comporter la définition de nouveaux types. Il s'agit en réalité de donner un nom à un ensemble de valeurs qui constituent le nouveau type. Une telle déclaration est une déclaration de synonymie. Ainsi, la déclaration

`typedef` *type identificateur* `;`

permet d'introduire un nouveau type nommé (*identificateur*) à partir d'un autre type, qui sera fourni par une description de type (indiquée ici par la catégorie *type*) introduite après le mot réservé `typedef`. N'ayant pas encore introduit les types dérivés, nous ne connaissons pas beaucoup de descriptions de type, mais les types de base nous permettent déjà de déclarer de cette manière

```
typedef int Dimension ;
typedef short int Position ;
```

On peut ainsi définir un type `Dimension` pour coder des largeurs et hauteurs de fenêtres, et un type `Position` pour stocker leur emplacement sur l'écran.

On a vu également qu'on pouvait introduire un type d'entiers restreints à partir d'une déclaration d'énumération de constante en lui donnant un nom, comme dans :

```
enum Couleur {VERT, BLEU, ROUGE, ORANGE} ;
```

La définition d'un type énuméré pourra alternativement être introduite explicitement par une déclaration avec `typedef` :

`typedef enum {` *liste_des_constantes* `}` *identificateur* `;`

Exemple :

```
typedef enum {VERT, BLEU, ROUGE, ORANGE} Couleur ;
Couleur teinte1, teinte2 = ROUGE ;
```

Ici, on a deux variables de type `Couleur`, la deuxième étant initialisée à `ROUGE`. (Le compilateur associe à chacune des constantes introduites par l'énumération une valeur entière, en commençant par zéro et en augmentant de 1 chaque valeur de constante suivante. Ainsi, la valeur de la constante VERT est `0`, celle de BLEU est `1`, celle de ROUGE `2`, etc.).

De manière générale, les déclarations de type avec `typedef` permettent de donner un nom à n'importe quel type dérivé (tableaux d'objets d'un type donné, fonctions retournant des objets d'un type donné, structures contenant des objets de divers types, etc.).

3.3.2 Structures

Une structure est une variable permettant de stocker une ou plusieurs variables de types quelconques. Contrairement au cas des tableaux, les valeurs rangées dans une structure ne sont pas nécessairement du même type. Les structures sont très utiles pour organiser les données, parce qu'elles permettent de traiter comme une seule entité un groupe de variables hétérogènes, mais conceptuellement reliées.

Un exemple classique est celui d'une feuille de paye qui pourra contenir différentes valeurs numériques correspondant au salaire versé, aux cotisations, ainsi que diverses chaînes de caractères fournissant la description de l'employeur et de l'employé (nom, adresse, etc.). D'autres exemples classiques proviennent de logiciel de dessins : un point sera défini par deux coordonnées, un rectangle par deux points, etc.

On accède aux différentes variables d'une structure à partir de noms arbitraires, appelés **champs** (ou **membres**) de la structure. La portée de ces noms est strictement restreinte à la structure elle-même. Pour accéder à un champ, on utilise l'opérateur d'accès, noté '.', sous la forme :

nom_de_structure`.`*nom_de_champs*

Exemple

Pour représenter les points du plan complexe en coordonnées polaires, on peut définir un type de structure nommé `Point`, et déclarer ensuite deux variables `origine` et `pt` de type `Point` comme ceci :

```
typedef struct {
```

```
        float angle;
        unsigned int  distance;
    } Point ;
    Point  origine, pt;
```

On peut ensuite initialiser les champs de la structure `origine` par

```
origine.distance = 0 ;
origine.angle = 0 ;
```

Les structures peuvent être imbriquées. Ainsi, un nouveau type de structure nommé `Rectangle` peut être introduit pour représenter un rectangle par deux points (situés en des coins diagonalement opposés) avec :

```
typedef struct {
      Point pt1 ;    /* coin supérieur gauche */
      Point pt2 ;    /* coin inférieur droit */
} Rectangle;
```

Pour accéder au champ `distance` du second membre d'une structure `rect` de type `Rectangle` on écrira

```
rect.pt2.distance
```

La déclaration en `typedef` n'est pas toujours nécessaire, et une alternative est d'écrire :

```
struct {
      float angle;
      unsigned int  distance;
} origine, pt ;
```

Cette tournure étant source d'erreurs et fastidieuse (elle doit être réécrite chaque fois qu'une variable de ce type est nécessaire !), le langage propose la définition d'un type structure par l'intermédiaire d'un *tag*. Un tag est un identificateur quelconque qui tient lieu de raccourci pour le bloc de déclaration des membres de la structure. Ainsi, dans l'exemple de points, on pouvait introduire le tag `tag_Point`, et définir un type de structure par :

```
struct tag_Point {
      float angle;
      unsigned int  distance;
};
```

Ces lignes constituent une *définition de type*, mais `tag_Point` n'est pas un identificateur de type. Ce n'est que précédé du mot `struct` qu'il fournit un type structure. Ainsi, on peut déclarer les variables `origine` et `point` par :

```
struct tag_Point  origine, pt ;
```

ou encore introduire le type `Rectangle` avec :

```
typedef struct {
      struct tag_Point pt1 ;   /* coin supérieur gauche */
      struct tag_Point pt2 ;   /* coin inférieur droit */
} Rectangle;
```

Mais la première manière de procéder que nous avons présentée est sans doute la plus claire, et nous l'adopterons pour l'instant. Signalons néanmoins que toute définition par tag peut être accompagnée d'une définition de type comme

```
typedef struct tag_Point Point ;
```

et que de telles définitions de types peuvent figurer *avant* la définition de la structure. L'intérêt de cette façon de procéder est que les définitions de structures imbriquées peuvent alors être introduites dans n'importe quel ordre[12].

3.3.3 Exemple de tableau

```
1   #include <stdio.h>
2
3   /* lit les caracteres entres un a un et compte le nombre
4      d'occurrences de chaque chiffre dans chaque categorie
5    */
6
7   int main ()
8   {
9     int c, i ;
10
11    int nbChiffre[10] ; /* nbChiffre[0] = le nb de 0 */
12                        /* nbChiffre[1] = le nb de 1 */
13          /* etc., jusqu'a nbChiffre[9] = le nb de 9 */
14
15    /* initialisation des cellules du tableau a zero */
16    i=0 ;
17    while (i < 10)
18    {
19         nbChiffre[i] = 0 ;
20         i = i + 1 ;
21    }
22    /* inspection un a un de chaque caractere lu */
23    /* pour compter chaque apparition d'un chiffre */
24    while ( (c = getchar()) != EOF)
25         if (c >= '0'  &&  c <= '9')
26              nbChiffre[c – '0'] = nbChiffre[c-'0'] + 1;
27
28    /* impressions des resultats du decompte */
29    i=0 ;
30    while (i < 10)
31         printf("%5d chiffre(s) %d\n",nbChiffre[i],i ++);
32
33    return 0 ;
34  }
```

Ce programme compte le nombre de caractères numériques (i.e. de chiffres `0`, `1`, ..., `9`) qui sont entrés à la console, et imprime en sortie le nombre de chiffres qui ont été lus dans chaque catégorie, c'est-à-dire, combien on a lu de zéros, de 1, etc.

Pour stocker le nombre de chiffres dans chaque catégorie, on utilise un type particulier que nous n'avons pas encore rencontré : le type tableau (ici un tableau d'entiers). Un tableau d'entiers est une structure portant un nom de variable unique, mais qui permet de stocker un

---

[12] et l'on pourra aussi, par ce procédé, définir des structures récursives. Nous introduirons cette notion plus loin.

ensemble de plusieurs valeurs. Le nombre de valeurs pouvant être stockées par un tableau doit être déclaré : c'est la taille du tableau. On accède aux différentes valeurs du tableau à l'aide d'un indice entier qui peut varier entre `0` et la taille du tableau diminuée de 1, et que l'on place entre crochets derrière le nom du tableau.

La déclaration

```
int nbChiffre[10] ;
```

déclare que la variable `nbChiffre` est un tableau de taille 10, et que les 10 valeurs stockées dans ce tableau seront des entiers de type `int`. Les valeurs sont rangées dans des zones mémoire (consécutives) numérotées, et pour accéder à une valeur, on utilisera un index entier entre crochets.

Ainsi `nbChiffre[0]` désigne la première zone mémoire du tableau, et `nbChiffre[9]` la dernière. Une telle expression se comporte comme un nom de variable et la déclaration d'un tableau de taille N permet en quelque sorte d'utiliser N nouvelles variables, appelées **cellules** du tableau. Les lignes 16 à 21 du programme servent à initialiser toutes les cellules du tableau à zéro.

```
i=0 ;
while (i < 10)
{
      nbChiffre[i] = 0 ;
      i = i + 1 ;
}
```

`nbChiffre[i]` est utilisé ici à gauche de l'opérateur d'affectation comme un identificateur de variable pour désigner la `i`ème cellule. On aurait pu réduire le nombre d'instructions du programme en utilisant l'opérateur d'incrémentation dans la boucle :

```
while (i < 10)
      nbChiffre[i++] = 0 ;
```

La fonction `getchar` utilisée dans l'instruction while suivante, lit un caractère sur le flot des entrées tapées au clavier de la console. Elle fait partie de la librairie d'entrée/sortie standard au même titre que `printf`, ce qui explique qu'il n'y ait qu'un seul fichier d'en-tête (`stdio.h`) inclus au début du programme. La fonction `getchar` retourne une valeur de type `int`, et c'est pourquoi, le « caractère » lu est déclaré ici comme entier et non comme `char`. Le type `char` est normalement utilisé pour les caractères, mais n'importe quel type entier peut en réalité convenir si l'on ne manipule pas de valeurs incongrues. La raison pour laquelle cette fonction utilise des `int` plutôt que des `char` est qu'un nombre négatif est renvoyé par la fonction pour signaler la fin (=l'absence) de caractères à lire sur le fichier des entrées. Il fallait en effet distinguer ce code particulier du code d'un caractère ordinaire.

La constante `EOF` est précisément définie dans le fichier `sdtio.h` par une valeur négative (EOF vaut `–1`). EOF signifie End Of File. Ce caractère de contrôle peut être émis au clavier sous UNIX par l'enfoncement simultané des touches Contrôle et D.

Le test figurant dans la boucle `while` est exécuté avant chaque nouvelle itération et il permet de réinitialiser la valeur de la variable `c` avec le code du caractère suivant lu sur la console. La boucle termine quand il n'y a plus de caractères à lire, c'est-à-dire quand le caractère « lu » vaut EOF.

Pour chaque caractère lu, on effectue les tests permettant de classer le caractère dans une des catégories que l'on souhaite dénombrer, et on incrémente la variable correspondante. Le programme termine par l'impression des résultats.

La ligne 26 mérite un commentaire. On parcourt tous les caractères entrés à la console en les lisant un à un avec `getchar` dans la condition du `while`. La variable `c` contient ici le code du dernier caractère lu, et si le test du `if` est franchi, c'est qu'il s'agit d'un chiffre. Dans ce cas, l'entier `c – '0'` est justement l'entier correspondant à ce chiffre, de par le fait que l'encodage des caractères ASCII code les caractères chiffres par des entiers successifs. La valeur indexée du tableau correspondant à ce chiffre est donc incrémentée, pour mémoriser qu'on a lu un chiffre de ce type. On aurait pu ici aussi utiliser l'opérateur d'incrémentation et écrire

```
if (c >= '0'  &&  c <= '9')
     ++nbChiffre[c – '0'];
```

La dernière boucle `while` réalise l'affichage du nombre de chiffres lus dans chaque catégorie, en parcourant successivement toutes les cellules du tableau. L'impression correspondant à l'entrée à la console du texte de ce programme serait :

```
12 chiffre(s) 0
 7 chiffre(s) 1
 0 chiffre(s) 2
 0 chiffre(s) 3
 0 chiffre(s) 4
 1 chiffre(s) 5
 0 chiffre(s) 6
 0 chiffre(s) 7
 0 chiffre(s) 8
 3 chiffre(s) 9
```

3.4 Autres opérateurs

3.4.1 Opérateurs binaires (bits à bits)

Les opérateurs binaires considèrent leurs opérandes (de type entier) comme des séquences de bits. Pour en comprendre le fonctionnement, il faut avoir quelques notions sur les représentations des nombres en machine (en particulier pour les entiers non signés, de leur forme binaire, en base 2). Nous ne les utiliserons donc pas pour l'instant et reverrons ces questions plus tard[13], après avoir étudié la manière dont sont représentés les nombres en machine. En voici simplement la liste, dans laquelle s, s1 et s2 sont des entiers, considérés donc pour leurs représentations comme suites de bits :

Opération	Nom	Signification
~ s	complémentation	Tous les bits de s sont inversés (0 →1 et 1→0)
s1 & s2	ET binaire	Un bit du résultat ne vaut 1 que si les bits correspondant dans s1 et s2 sont tous les deux à 1

[13] Ces opérateurs sont très utilisés pour définir des « masques » permettant de tester rapidement l'appartenance multiple à diverses catégories de données.

s1 | s2	OU binaire	Un bit du résultat vaut 1 si un bit correspondant dans s1 ou s2 est à 1
s1 ^ s2	OU exclusif	Un bit du résultat ne vaut 1 que si les bits correspondant dans s1 et s2 sont différents
s1<<s2	décalage à gauche	Les bits du résultat sont ceux de s1 décalés à gauche de s2 positions, les nouvelles à droite valant 0
s1>>s2	décalage à droite	Les bits du résultat sont ceux de s1 décalés à droite de s2 positions, les nouvelles à gauche valant 0

3.4.2 Opérateur `sizeof`

Cet opérateur est un opérateur (unaire) qui porte sur un type ou une expression. Il retourne la taille en octets de la zone mémoire nécessaire pour stocker une valeur du type de son opérande. Cette valeur de retour est un entier non signé. Elle est déterminée à la compilation et non à l'exécution et est donc assimilable à une constante.

Par définition, `sizeof(char) == 1`, de même que `sizeof(signed char)` et `sizeof(unsigned char)` . Sur une machine où les `int` sont représentés sur 2 octets, pour une variable `varInt` de type `int`, on aura de même `sizeof(varInt) == 2`.

3.4.3 Opérateur conditionnel

L'opérateur conditionnel est un opérateur à trois opérandes (ternaire) qui sont des expressions. Le premier opérande est une expression qui sera évaluée comme expression booléenne, notée ici *expression_conditionnelle*, pour en souligner le rôle déterminant :

expression_conditionnelle **?** *expression1* **:** *expression2*

L'expression conditionnelle (premier opérande) est d'abord évaluée. Si sa valeur est VRAI, l'opérateur calcule et retourne alors la valeur de la première expression (deuxième opérande) et le dernier opérande (*expression2*) n'est pas évalué. Dans le cas contraire, c'est le second opérande (*expression1*) qui n'est pas évalué, et l'opérateur calcule et retourne la valeur du dernier opérande (*expression2*).

Par exemple, on pourra utiliser l'opérateur conditionnel

```
(a >= b) ? a : b
```

pour calculer le maximum de a et b. Les parenthèses autour de l'expression conditionnelle ne sont pas obligatoires, mais nous recommandons d'en mettre systématiquement pour clarifier la lecture des trois opérandes.

Cet opérateur peut être utile pour faire figurer une alternative dans une expression (par exemple dans une boucle `for`). Mais il est surtout utilisé avec la directive `#define` qui s'adresse au préprocesseur pour écrire des « macro-fonctions », qui sont en réalité des expressions complexes, obtenues par substitution textuelle.

3.4.4 Opérateur de séquence

L'opérateur de séquence ',' est utilisé quand les règles de syntaxe (la grammaire) du langage imposent d'écrire une seule expression, mais que le traitement à effectuer nécessite l'évaluation de plusieurs expressions. Les expressions séparées par l'opérateur de séquence

sont effectuées de gauche à droite, les unes après les autres, et le résultat final est celui de la *dernière* expression.

```
(max  < x) ?  (echange = x, max = x,  x = echange) : max
```

Ici, les valeurs de `x` et `max` sont éventuellement permutées (à l'aide de la variable auxiliaire `echange`) pour que `max` contienne toujours la plus grande valeur ; cette expression retourne dans tous les cas la plus grande valeur, rangée dans `max`.

3.4.5 Divers

Les opérateurs restants seront présentés plus loin. Il s'agit de l'appel de fonction (noté par deux parenthèses), l'indexation dans un tableau, noté par deux crochets `[]` et la sélection d'un champ dans une structure, noté par un point. L'opérateur d'adresse `&` et les opérateurs `*` et –› seront présentés dans le chapitre sur les pointeurs.

3.4.6 Précédence et associativité des opérateurs

La Figure 5 fournit la table de l'associativité de tous les opérateurs, énumérés par ordre de priorité décroissante. Cette table complète celle de la Figure 3 qui ne présente que les opérateurs numériques et logiques.

Les opérateurs les plus prioritaires sont ceux des premières lignes de la table. La première ligne comporte l'appel de fonction, l'indexation de tableau, et les opérateurs de sélection de champs dans des structures. La seconde la négation logique et la négation binaire, les opérateurs d'incrémentation et décrémentation, les signes unaires, le déréférençage, l'opérateur d'adresse, la conversion de type forcée et l'opérateur `sizeof`. Viennent ensuite les opérations numériques (lignes 3 et 4), les opérations de décalage bit-à-bit (ligne 5), les opérateurs de comparaisons (lignes 6 et 7), les opérateurs binaires (lignes 8, 9 et 10), puis les opérateurs logiques (lignes 11 et 12). Tout en bas, l'opérateur conditionnel (ligne 13), les affectations (ligne 14) et l'opérateur de séquencement.

	Opérateurs	Associativité
1	( ) [ ] → .	gauche à droite
2	! ~ ++ — + – * & (*type*) `sizeof`	droite à gauche
3	* / %	gauche à droite
4	+ –	gauche à droite
5	<< >>	gauche à droite
6	< <= > >=	gauche à droite
7	== !=	gauche à droite
8	&	gauche à droite
9	^	gauche à droite
10	|	gauche à droite
11	&&	gauche à droite
12	||	gauche à droite
13	? :	droite à gauche
14	= += –= *= /= %= &= ^= |= <<= >>=	droite à gauche
15	,	gauche à droite

Figure 5
Précédence et associativité des opérateurs

4 Les instructions du langage

4.1 Instruction et blocs d'instructions

Les instructions servent à indiquer un traitement à faire à l'ordinateur. Ce traitement consiste pour une instruction de base, en le calcul d'une expression qui a un effet de modification (une affectation), ou en un appel de fonction (par exemple, une fonction d'entrée/sortie).

Les instructions sont regroupées par blocs dans lesquels l'ordre d'exécution des instructions est séquentiel (on passe à l'instruction suivante dans l'ordre où elles sont écrites dans le programme), et il existe des instructions de contrôle, combinant plusieurs instructions, qui permettent d'organiser un ordre d'exécution différent de l'ordre standard.

Une instruction est formée à partir d'une simple expression en lui ajoutant un point-virgule. Par exemple :

```
max = a * b + 5 ;
i++;
printf("Bonjour tout le monde !\n");
```

Le point-virgule fait partie de l'instruction (il la termine). Ce n'est pas en C un séparateur d'instructions, contrairement à beaucoup d'autres langages (comme Pascal).

Le langage C autorise n'importe quelle expression à être transformée en instruction par ajout d'un point-virgule, mais nous recommandons de limiter les instructions aux expressions dont le dernier opérateur exécuté est un opérateur de modification, ou aux appels de fonctions qui retournent le type `void`.

Dans quelques cas (rares), les règles du langage demandent une instruction et il se trouve qu'il n'y a aucun traitement à effectuer. On peut alors utiliser le point-virgule seul, qui tient lieu d'**instruction vide**.

Un **bloc d'instructions** est une séquence d'instructions encadrées par des accolades. Il s'agit en réalité d'une instruction particulière, ou **instruction composée**, pouvant apparaître à la place de n'importe quelle instruction (généralement à l'intérieur d'une instruction de contrôle), quand le traitement nécessaire ne peut s'écrire avec une seule instruction. Les fonctions sont également définies par un bloc d'instructions que l'on appelle le corps de la fonction. Les programmes les plus simples ne contiennent qu'un seul bloc d'instructions : le corps de la fonction `main`.

La forme générale d'un bloc d'instructions est :

```
{
        déclaration(s)

        instruction(s)
}
```

Les déclarations sont optionnelles. On peut également avoir un bloc sans instructions ni déclarations. Il s'agit alors du **bloc vide** (analogue à l'instruction vide **' ;'**). Des déclarations figurent généralement en tête du corps des fonctions. Elles sont ainsi facilement identifiables. Pour plus de clarté, on les séparera du début des instructions en ajoutant une ligne blanche.

Les variables définies par ces déclarations ne sont « visibles » (ou « connues ») qu'à l'intérieur du bloc (et éventuellement des sous-blocs inclus). On dit que la **portée** de leur définition est le bloc. En règle générale, on choisit la plus petite portée pour une déclaration, afin de ne pas saturer l'esprit du programmeur de noms de variables. Mais il est raisonnable aussi de ne pas introduire trop de zones différentes pour les déclarations de variables, et nous recommandons aux débutants d'éviter de déclarer des variables à l'intérieur des sous-blocs d'instructions figurant dans les instructions de contrôle.

Les différents blocs et sous-blocs figurant dans des instructions composées seront présentés avec un décalage systématique vers la droite : l'indentation. On positionnera en particulier avec soin l'accolade fermante du bloc pour qu'elle puisse facilement être identifiée comme correspondant à ce bloc d'instructions.

Si des variables de même nom sont utilisées dans plusieurs sous-blocs, la variable connue par le bloc le plus interne est celle qui a été déclarée la plus récemment. Ainsi,

```
{
        int i ;
        s1
        {
             int j ;
             s2
             {
                  float i ;
                  s3
             }
        }
}
```

En s1, la variable `i` est connue comme entier de type `int`. Mais la variable `j` est inconnue. En s2, les variables `i` et `j` sont connues comme entiers de type `int`. Par contre, en s3, la variable `j` reste connue, mais la variable `i` de type `int` référencée dans les deux blocs précédents est « masquée » par la définition d'une nouvelle variable `i`, de type `float`, (inconnue ailleurs). Si l'on définissait un nouveau bloc interne à s3, une référence à `i` dans ce dernier bloc le plus interne désignerait la variable `i` de type `float`, et non pas celle définie comme `int` dans le bloc le plus englobant.

L'indentation sert ici à souligner la portée des variables. On a déjà vu qu'elle sert aussi à souligner la (ou les) instruction(s) interne(s) dans une instruction de contrôle, comme les instructions conditionnelles ou les boucles.

4.2 Instructions de contrôle

Les instructions de contrôle sont des instructions qui permettent de contrôler et d'orienter l'exécution (en faisant des tests sur les données). On les appelle aussi parfois instructions de conduite, parce qu'elles « conduisent » l'exécution du programme.

4.2.1 L'instruction conditionnelle

Formellement, la syntaxe d'une instruction conditionnelle est

```
if  ( expression )
        instruction1
else
```

instruction2

où la seconde partie commençant par `else` est optionnelle. L'expression qui figure entre parenthèses est calculée comme expression booléenne (c'est-à-dire considérée comme VRAI si elle est non nulle). On peut donc écrire aussi bien

```
if  ( expression )
```

que

```
if  ( expression != 0)
```

et on choisira la tournure qui parait la plus claire. Cette première expression est toujours calculée. Si elle est VRAI (c'est-à-dire si elle a une valeur non nulle), *instruction1* est exécutée. Si elle est fausse (*expression* vaut zéro) et s'il existe une partie commençant par `else`, *instruction2* est exécutée et *instruction1* est ignorée.

Le fait que la partie en `else` soit facultative introduit une ambiguïté quand *instruction1* est elle-même une instruction `if` conditionnelle :

```
if (n > 0)
      if (a > b)
            z = a ;
      else
            z = b ;
```

Ici, on a marqué par l'indentation que le `else` se raccrochait au deuxième `if`, mais en réalité, le langage n'a pas la notion d'indentation. Cependant c'est effectivement comme cela que le compilateur interprètera ces instructions, parce que pour résoudre l'ambiguïté, une partie en `else` se raccroche toujours au `if` précédent. Si l'on souhaite avoir l'autre interprétation, on utilisera des accolades :

```
if (n > 0)
{
      if (a > b)
            z = a ;
}
else
      z = b ;
```

Une conséquence des règles ci-dessus permet la construction classique suivante qui mérite d'être mentionnée. Il s'agit de l'enchaînement d'instructions conditionnelles figurant toujours dans la seconde partie (en `else`) de la conditionnelle précédente. C'est une manière d'écrire des cas disjoints. Cet enchaînement de conditionnelles peut être indenté de la manière suivante :

```
if (expression)
      instruction
else if (expression)
      instruction
else if (expression)
      instruction
else if (expression)
      ...
```

```
else
```
instruction

Les expressions booléennes sont testées dans leur ordre d'apparition, et si l'une d'entre elle est VRAI, l'instruction associée est exécutée et termine toute la chaîne. Comme toujours, chaque instruction peut être un bloc d'instructions (instruction composée) figurant entre accolades. Le dernier cas

```
else
```
instruction

peut être omis.

Exemple

```
#include <stdio.h>

/* ce programme inspecte chaque caractère lu et compte ceux
  qui sont des chiffres, des blancs, ou differents */
int main ()
{
      int  c, i,
           nbChiffres, nbBlancs, nbAutres ;

      nbChiffres = nbBlancs = nbAutres = 0 ;

      while ( (c = getchar()) != EOF)
           if (c >= '0'  &&  c <= '9')
                nbChiffres++ ;
           else if (c == ' ' || c == '\n' || c == '\t')
                nbBlancs++ ;
           else
                nbAutres++ ;

      printf("chiffres = %d ", nbChiffres );
      printf("blancs = %d ", nbBlancs );
      printf("Autres = %d\n",nbAutres);
      return 0 ;
}
```

On a déjà rencontré la fonction `getchar` et une boucle `while` similaire dans le programme donné dans la section 3.3.3 comme exemple de tableau .

Pour chaque caractère lu, on effectue les tests permettant de classer le caractère dans une des catégories que l'on souhaite dénombrer, et on incrémente la variable correspondante. Le programme se termine par l'impression des résultats.

Exercice

Ecrire une variante du programme précédent qui compte le nombre de caractères alphanumériques, le nombre de lignes et le nombre de « mots » entrés au clavier - un mot étant une séquence de caractères quelconques séparée du reste par des caractères blancs (un nombre sera donc considéré comme un mot).

4.2.2 L'instruction de branchement (`switch`)

L'instruction de branchement ou `switch`, permet de comparer la valeur d'une expression avec celles de constantes entières, listées dans des cas séparés. Si la valeur coïncide avec une de ces constantes (testées dans l'ordre), on réalise un « branchement » sur les instructions qui suivent dans le `switch`. La syntaxe d'une instruction `switch` est la suivante :

```
switch ( expression )
{
      case constante1 : instructions
      case constante2 : instructions
           ...
      case constanteN : instructions
      default : instructions
}
```

Le branchement consiste à poursuivre l'exécution en enchaînant sur les instructions qui débutent au niveau du cas, puis sur celles qui figurent dans les cas suivants, jusqu'à la rencontre d'une instruction `break`, qui interrompt cette exécution et termine le `switch`. S'il n'y a pas d'instruction `break`, toutes les instructions suivantes sont exécutées, y compris celles du cas par défaut, et on quitte le `switch`. La partie `default` est optionnelle et permet de réaliser un branchement sur un groupe d'instructions finales quand aucune des constantes mentionnées au-dessus ne coïncide avec la valeur de l'expression. Le groupe d'instructions correspondant à un cas peut être vide, auquel cas l'exécution enchaîne sur le groupe d'instructions coïncidant avec le cas suivant.

Ainsi, le programme précédent pouvait s'écrire :

```
#include <stdio.h>

/* compte le nombre de caracteres lus comme chiffres,
      blancs ou autres */
int main ()
{
      int  c, i,
           nbChiffres, nbBlancs, nbAutres ;

      nbChiffres = nbBlancs = nbAutres = 0 ;

      while ( (c = getchar()) != EOF)
      {
           switch (c)
           {
                case '0' :
                case '1' :
                case '2' :
                     ...etc .
                case '9' :     nbChiffres++ ;
                               break;
                case ' ' :
                case '\n' :
                case '\t' :    nbBlancs++ ;
                               break;
```

```
                default :       nbAutres++ ;
                                break ;
            }  /* fin du switch */
        } /* fin du while */

        printf("chiffres = %d ", nbChiffres );
        printf("blancs = %d ", nbBlancs );
        printf("Autres = %d\n",nbAutres);

        return 0 ;
}
```

La plupart du temps, on termine une série d'instructions de branchement sur un cas par une instruction `break`, qui termine l'instruction `switch` (et évite d'enchaîner sur les instructions des cas suivants). Une instruction `break` interrompt l'instruction englobante en cours, et permet ici de sortir du `switch`. On verra qu'on peut aussi l'utiliser pour forcer la sortie d'une boucle `while` ou d'une boucle `for`.

4.2.3 Boucle `while` et boucle `for`

Nous avons déjà rencontré l'instruction `while`. Dans

`while (` *expression* `)`
instruction

l'*expression* est évaluée. Si elle est VRAI (c'est-à-dire non nulle), l'*instruction* est exécutée puis l'*expression* est ré-évaluée. Ce cycle se perpétue jusqu'à ce que l'expression soit fausse, c'est-à-dire qu'elle prenne la valeur zéro (ce qui peut d'ailleurs se produire dès le début, auquel cas, on ne pénètre jamais dans le corps de la boucle). L'*instruction* n'est alors pas exécutée, et la boucle `while` termine.

L'instruction `for` du C est une généralisation de la boucle `while` :

`for (` *expression1* ; *expression2* ; *expression3* `)`
instruction

Elle est exactement équivalente à

expression1 ;
`while (` *expression2* `)`
`{`
instruction
expression3 ;
`}`

Elle permet donc d'indiquer deux choses supplémentaires : une instruction à exécuter une première (et unique) fois avant de tenter de pénétrer dans la boucle (l'instruction *expression1* ;), et une instruction supplémentaire (l'instruction *expression3* ;) à effectuer systématiquement en fin de cycle d'itération, *avant* de reprocéder au test (*expression2*) autorisant une nouvelle exécution du corps de la boucle.

Du point de vue grammatical, *expression1*, *expression2* et *expression3* sont des expressions. Mais le plus souvent, *expression1* et *expression3* seront des affectations, et *expression2* une

expression booléenne (exprimant une relation). Toutes ces expressions peuvent être omises, mais pas les points-virgules. Ainsi

```
for (   ;   ;  )
      ...
```

instaure une boucle potentiellement infinie, dont on sortira probablement par une instruction `break`, placée quelque part dans le corps d'exécution de la boucle. Une boucle `while` est donc directement équivalente à

```
for (    ; expression ;  )
      instruction
```

Utiliser une boucle `while` ou une boucle `for` est principalement une affaire de goût. Mais s'il n'y a ni initialisation, ni ré-initialisation, comme dans

```
while ( (c = getchar()) == ' ' || c == '\n' || c == '\t')
      ;     /* instruction vide: il s'agit juste ici de
               "manger" les caracteres blancs ou séparateurs */
```

la boucle `while` sera plus naturelle. Par contre, la tournure idiomatique suivante est très caractéristique du langage C :

```
for (i = 0 ; i < n ;  i++)
      ...
```

Elle permet en particulier de parcourir les n élements d'une chaîne de caractères, ou d'un tableau. Le programme de la section 3.3.3 peut s'écrire plus simplement avec des boucles `for`.

```
1   #include <stdio.h>
2
3   /* compte le nombre d'occurrences de chaque
4      categorie de chiffres  */
5
6   int main ()
7   {
8     int c, i ;
9
10    int nbChiffre[10] ; /* nbChiffre[0] = le nb de 0 */
11                        /* nbChiffre[1] = le nb de 1 */
12          /* etc., jusqu'a nbChiffre[9] = le nb de 9  */
13
14          /* initialisation du tableau */
15    for (i=0 ; i < 10 ;  ++i)
16          nbChiffre[i] = 0 ;
17
18    while ( (c = getchar()) != EOF)
19          if (c >= '0'  &&  c <= '9')
20               ++nbChiffre[c – '0'];
21
22          /* impression des resultats */
23    for (i=0 ; i < 10 ;  ++i)
24          printf("%5d chiffre(s) %d\n", nbChiffre[i], i );
25
26    return 0 ;
```

```
27  }
```

Exercice

Ecrire une version du programme de conversion Fahrenheit-Celsius avec une boucle `for` au lieu d'une boucle `while`.

4.2.4 Boucle `do-while`

Comme nous venons de le voir, les boucles `while` et `for` testent la condition indiquée dans l'expression avant chaque nouvelle entrée dans la boucle. La boucle `do-while` au contraire, effectue ce test *après* exécution du corps de la boucle ; le corps de cette boucle est donc toujours effectué au moins une fois (alors que celui d'une boucle `while` ou d'une boucle `for` peut être ignoré si la condition n'est pas VRAI initialement).

La syntaxe d'une instruction `do-while` est la suivante :

```
do
        instruction
while   ( expression ) ;
```

L'instruction (ou le bloc d'instructions) est exécutée, puis l'expression est évaluée. Si elle est VRAI, l'instruction est exécutée à nouveau, et ainsi de suite. Quand l'expression devient fausse, la boucle termine.

L'expérience montre que la boucle `do-while` est beaucoup moins utilisée que les boucles `while` et `for`. (Elle est équivalent à l'instruction `repeat-until` de Pascal).

4.2.5 Instructions de rupture de séquence

Ces instructions font que l'instruction suivante exécutée n'est pas celle qui suit textuellement, mais une autre qui dépend de l'instruction de rupture de séquence. Elles sont souvent pratiques, mais nuisent à la lisibilité du programme et sont à utiliser avec précaution.

L'instruction `break` ne peut figurer que dans des boucles ou une instruction `switch` de branchement, et elle permet de court-circuiter la fin de l'exécution de ces instructions. (Dans une instruction `switch`, contrairement à ce que nous venons de dire, les `break` clarifient plutôt le texte du programme et nous recommandons d'en utiliser un après chaque séquence d'instructions associées à une constante particulière).

A l'intérieur d'une boucle `for`, `while` ou `do-while`, l'instruction `break` permet de sortir immédiatement. Si plusieurs boucles sont imbriquées, le programme quitte la boucle la plus interne.

L'instruction `continue` ressemble en partie à l'instruction `break`, mais elle ne s'applique qu'aux boucles et pas aux `switch`. Elle est relativement peu utilisée. Son effet est d'interrompre la boucle d'itération en cours et de passer à l'itération suivante (c'est-à-dire au test permettant ou non une nouvelle exécution du corps de la boucle). Une instruction `continue` placée dans un `switch` lui même situé dans une boucle aura pour effet d'interrompre le `switch` et de passer au test d'itération de boucle suivant.

Le langage C possède aussi une instruction de branchement permettant d'enchaîner sur n'importe quelle autre instruction du programme (indiquée dans le texte par une étiquette). Cette instruction intitulée `goto` a mauvaise réputation, car elle masque la structure logique du programme. Il y a cependant quelques cas particuliers d'abandon de boucles imbriquées où

elle peut trouver sa place, mais on peut toujours s'en passer, et nous ne l'utiliserons pas dans ce cours.

4.3 Instructions d'entrée/sortie

Les instructions d'entrées/sorties en C ne sont pas spécifiques au langage, et elles sont réalisées par des appels de fonctions prédéfinies dans des librairies. Nous en présentons ici quelques unes, qui font partie de la librairie d'entrée/sortie standard `stdio`, mais il en existe beaucoup d'autres.

4.3.1 Entrée/sortie de caractères : `getchar` et `putchar`

Il existe deux fonctions permettant de lire ou d'écrire des caractères (un à un) sur l'entrée et la sortie standard. Il s'agit des fonctions :

`getchar()` qui lit un caractère saisi au clavier et en retourne la valeur (un `int`).
`putchar(c)` qui affiche un caractère sur la console et en retourne la valeur.

Ces fonctions permettent en particulier de saisir des lignes comportant des caractères d'espacement. Leurs appels peuvent être mélangés à ceux de `printf` et `scanf`.

Exemple

```
#include <sdtio.h>
/* copie l'entrée standard vers la sortie standard */
int main ()
{
      int c ;

      c = getchar() ;
      while (c != EOF)
      {
           putchar(c);
           c = getchar();
      }
}
```

Ce programme permet de recopier tout caractère entré au clavier sur l'écran. Voici une variante classique qui compte le nombre des lignes entrées :

```
#include <sdtio.h>

/* compte et imprime le nb de lignes entrées */
int main ()
{
      int c, nbLignes = 0;

      while ( (c = getchar()) != EOF)
           if (c == '\n')
                nbLignes++ ;
      printf("nb de lignes = %d\n", nbLignes) ;
}
```

4.3.2 Entrée/Sortie « formatées » : `Printf` et `scanf`

Nous avons déjà présenté les fonctions `printf` et `scanf` sur des exemples. Ces fonctions **variadiques** (c'est-à-dire pouvant prendre un nombre arbitraire de paramètres) retournent une valeur entière, mais celle-ci est généralement ignorée. (La fonction `printf` retourne le nombre de caractères qui sont imprimés sur la sortie standard, c'est-à-dire sur l'écran de la console depuis laquelle on a lancé le programme ; et `scanf` retourne le nombre de valeurs qui ont été lues et rangées dans des variables).

Le premier paramètre de ces deux fonctions est obligatoire. C'est une chaîne de caractères qui peut indiquer, via des descripteurs, comment interpréter les paramètres suivants. La forme générale d'un descripteur commence par un caractère %, et est suivi (ou non) de signes optionnels précisant par exemple un nombre minimal (ou maximal) de caractères devant être respectivement imprimés (ou lus) ; le descripteur termine toujours par une partie obligatoire, le **spécificateur de type**, qui indique le type de la valeur à lire ou à imprimer. Le tableau ci-dessous fournit les spécificateurs des types de base.

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`

Signalons aussi l'existence du spécificateur `%s` qui permet d'imprimer un tableau (ou chaîne) de caractères avec `printf` (ce type sera présenté plus loin). Le spécificateur `%s` permet aussi de ranger, avec `scanf`, une séquence de caractères dans un tableau de caractères, mais dans le cas de `scanf`, la séquence de caractères lus ne pourra pas contenir de caractères blancs.

Exemple

```
1 #include <stdio.h>
2
3 /* Ce programme divise deux valeurs reelles saisies au
   clavier et imprime le resultat de la division */
4
5 int main () {
6     double r1,r2 ;
7
8     printf("entrez deux reels : ");
9     scanf("%lf%lf", &r1, &r2);
10
11    if (r2 != 0)
12        printf("Division = %lf\n", r1/r2);
13    else
14        printf("On ne peut pas diviser par 0 \n");
15    return 0 ;
16 }
```

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 entre les valeurs 4.5 et 3.215, l'affichage à l'écran sera :

entrez deux reels : **4.5 3.215**
Division = 1.399689

Remarque complémentaire : ces fonctions n'interagissent pas en réalité directement avec le clavier ou l'écran. Elles travaillent toujours avec une mémoire intermédiaire appelée **tampon** (*buffer*) qu'elles explorent avec un index. (Ces mémoires et ces index seront partagés par les autres fonctions opérant sur ces entrée et sortie standard). On peut alors avoir des surprises sur le comportement de ces fonctions qui est parfois délicat.

En particulier avec `printf`, les caractères sont susceptibles de ne pas être imprimés au moment souhaité. L'impression à l'écran ne s'effectue en effet que lorsque le tampon intermédiaire est plein. Il est alors « vidé » sur l'écran. L'impression d'un retour à la ligne a pour effet de vider le tampon, c'est pourquoi il est recommandé de terminer une série d'impression avec `printf` par l'impression d'un '`\n`'. On sera alors assuré que l'impression sur l'écran a bien eu lieu au moment souhaité par le programme.

Avec `scanf`, la situation est parfois encore plus délicate. Quand le tampon est vide (en particulier, lors de la première utilisation de scanf dans le programme), une demande de lecture du clavier est demandée. Les caractères sont alors lus et mis dans le tampon, jusqu'à ce que l'utilisateur frappe un retour à la ligne. Ce caractère d'activation est mis en bout du tampon, et l'index positionné au début. Au fur et à mesure des lectures avec `scanf`, l'index progresse. Si cet index arrive en bout de tampon, une nouvelle demande est adressée au clavier, et l'on repart avec un nouveau tampon et l'index est à nouveau positionné au début.

D'autre part, les indications de format fournies par `scanf` lui font ignorer les blancs initiaux avant la lecture requise. Il lit ensuite le nombre (ou la chaîne de caractères) attendu(e), mais peut arrêter la progression de l'index en cours de route si le nombre maximal de caractères est atteint. Au cas où un descripteur de format ne peut permettre de lire les caractères attendus sans erreur, `scanf` s'arrête, et l'on risque dans les appels suivants de rester bloqué[14]. Les lectures avec `scanf` demandent donc un formatage très précis des caractères entrés par l'utilisateur au clavier, et il n'est pas toujours facile de prévoir tous les cas.

Les descripteurs de type qui terminent obligatoirement une spécification de format de `scanf` sont les mêmes que ceux de `printf` (à quelques nuances près). `scanf` accepte aussi deux descripteurs spécifiques constitués de crochets ouvrant et fermant pour lire une chaîne de caractères[15]. Un autre descripteur utile particulier à `scanf` est l'étoile, qui indique de lire d'abord une valeur sans la ranger, ce qui revient donc à la sauter.

---

[14] Une solution dans ce type de situation est parfois de sauter le caractère gênant la lecture en lisant un caractère sans l'interpréter (par exemple avec `getchar`).

[15] Les crochets permettent de spécifier, soit un ensemble de caractères auquel doivent appartenir tous les caractères de la chaîne lue, soit un ensemble de caractères interdits (i.e. qui ne doivent pas figurer dans la chaîne). La chaîne lue est alors la plus longue chaîne possible (jusqu'au retour à la ligne) satisfaisant ces conditions.

Exemple

```
scanf("%u ,%20s ,%*20s%d%d", &numero, chaine, &val1, &val2)
scanf(" %c", &reponse)
```

Le premier appel saisira une ligne de la forme suivante :
(blancs)/entier/blancs/virgule/(blancs)/mot/blancs/virgule/(blancs)/
mot (ignoré)/(blancs)/entier/blancs/entier
dans laquelle « blancs » est une suite (éventuellement vide si entre parenthèses) de caractères blancs (espace, tabulations, nouvelle ligne, nouvelle page) et « mot » une suite quelconque d'au plus 20 caractères non blancs. Les variables `val1` et `val2` seront de type `int`, et `numero` de type `unsigned int`. La variable `chaine` aura le type tableau de caractères `char[21]` et sera de taille 21 car un caractère supplémentaire est nécessaire pour indiquer la fin des caractères du tableau (voir plus loin). Le dernier mot avant lecture des deux derniers entiers sera simplement ignoré.

La spécification d'un espace blanc dans la chaîne de formatage impose la présence d'au moins un caractère « blanc » dans la chaîne des entrées. De même la virgule impose la présence d'une virgule. Ainsi, la demande de lecture d'un caractère du second exemple nécessite la présence d'un (ou plusieurs) caractères blancs avant attribution d'une valeur à la variable `reponse` de type `char`. Si l'on ne met pas d'espace blanc dans la chaîne de format, les caractères blancs qui précèdent une entrée seront considérés comme un séparateur unique et ignorés.

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.