Initiation au C

Éric Brunet (Eric.Brunet@lps.ens.fr)

12 février 2007

Introduction

Le langage C a été inventé au début des années 1970 par Ken Thompson et Dennis M. Ritchie, les développeurs du système d'exploitation UNIX. Les buts du C étaient d'avoir un langage suffisamment proche des opérations qu'un processeur était capable d'accomplir, ce qui permettait de faire assez simplement des compilateurs efficaces, et de suffisamment haut niveau pour qu'il soit possible d'écrire rapidement des programmes capables de tourner sur des machines différentes. Pour ces raisons, le C a été décrit comme un « assembleur portable. »

Le C s'est inspiré de langages déjà existants (BCPL, Fortran, Algol) et a été en évolution constante. À un moment donné, un certain compilateur C pouvait accepter des constructions qu'un autre compilateur refusait et, plus grave, il est arrivé que plusieurs compilateurs du C interprètent de manière différente la même construction.

Pour que tout le monde soit d'accord sur ce qu'était effectivement le C, il était important d'avoir une description de référence du langage, c'est-à-dire une norme. Ce rôle a longtemps été joué par le livre *The C Programming Language*, de Brian Kernighan et Dennis M. Ritchie, publié en 1978. Le langage décrit dans ce livre est connu sous le nom de « C K&R. » Le consortium ANSI (*American National Standards Institute*) a mis six ans pour écrire une norme officielle, publiée en 1989 et acceptée un an plus tard le consortium ISO (*International Organization for Standardization*). Le langage décrit dans ces documents est connu sous le nom de « C ANSI » ou « C 89 », ou « C ISO » ou « C 90 ». La dernière mouture de la norme, approuvée en 1999, est connue sous le nom de « C 99. » Il y a cependant encore assez peu de compilateurs qui acceptent toutes les constructions du C 99.

Malgré tout ce travail de normalisation, toutes ces définitions officielles du C n'ont jamais empêché la prolifération de diverses extensions du langage propres à chaque compilateur, mettant ainsi en danger la portabilité du C qui avait fait son succès. C'est pour ces raisons qu'il y a eu plusieurs normes du langage C : à chaque nouvelle version, les comités de normalisation ont examiné les pratiques existantes et les extensions au langage C présentes dans les compilateurs de l'époque, et ont déterminé, à partir de considérations pratiques, lesquelles de ces extensions devaient être introduites au langage et sous quelles formes. Mais, à chaque fois, l'un des principaux objectifs des comités a été de faciliter la transition et de troubler le moins possible la vaste collection de programmes C déjà existants ; si la norme recommandait une nouvelle manière de faire, l'ancienne manière était toujours acceptée et il y a de fortes chances qu'un programme écrit en C K&R soit toujours compréhensible par un compilateur récent.

Le langage présenté dans ce cours est essentiellement un sous ensemble du C ANSI avec quelques emprunts au C 99. Certaines constructions archaïques (comme par exemple

les déclarations K&R des fonctions), bien que légales en C ANSI afin de permettre la compilation de vieux programmes, seront bannies de ce cours au profit des constructions plus modernes qui sont à la fois plus efficaces, plus claires et plus sûres. En ce qui concerne les quelques constructions que nous utiliserons et qui sont autorisées par C 99 mais pas par C ANSI, comme les commentaires introduits par // ou les déclarations de variables locales au milieu d'une fonction, leur choix a été motivé par les services qu'elles rendaient et par le fait que ces constructions étaient acceptées en tant qu'extensions par la majorité des compilateurs avant d'être normalisées.

D'un autre côté, ce cours ne décrit pas complètement le langage C ; de nombreux aspects importants du C, comme les allocations dynamiques de la mémoire, ne sont pas traités ici. La partie du C qui est présentée ici permet déjà, néanmoins, de faire des programmes intéressants et donc de s'initier à la programmation.

Les deux premiers chapitres de ce cours sont les plus importants. Le premier chapitre contient un aperçu rapide et fondé sur des exemples simples de tout ce qu'il faut savoir pour pouvoir écrire ses premiers programmes, et le deuxième chapitre donne des conseils sur la manière de corriger les erreurs que l'on commet presque inévitablement en écrivant un programme. La fin du deuxième chapitre est un bon endroit pour arrêter la lecture de ce cours et pour s'installer devant un ordinateur et faire ses premiers essais. À partir du chapitre 3, ce cours rentre plus dans les détails et essaye de décrire plus systématiquement divers aspects du C.

Tout au long de ce cours, les paragraphes écrits en retrait et en petit contiennent des remarques, des conseils ou des informations sur le C qui sont plus compliqués que ce qui est écrit dans une taille normale. Ces paragraphes peuvent (et, sans doute, doivent) être ignorés lors d'une première lecture.

Chapitre 1

Structure d'un programme C

1.1 Fichier source et compilation

Dans la situation la plus simple, un programme C se présente sous la forme d'un *fichier source* écrit dans le format « texte » et dont le nom doit obligatoirement se terminer par « .c ». Ce source contient des déclarations de variables et les définitions d'une ou de plusieurs fonctions. Une de ces fonctions doit s'appeler `main()` ; c'est le point d'entrée du programme : quand le programme est lancé, les instructions qui composent la fonction `main()` sont exécutées dans l'ordre, et le programme se termine quand la fonction `main()` se termine. Toute fonction, et en particulier la fonction `main()`, peut appeler une ou plusieurs fonctions définies dans le fichier source ou dans des *bibliothèques externes*.

Un programme C peut en fait être composé de *plusieurs* fichiers sources. Dans ce cas, seul l'un de ces fichiers contient une fonction `main()`. Quand un programme devient trop gros, il est presque indispensable, pour des raisons de maintenance et de réutilisabilité, de le séparer en plusieurs fichiers sources indépendants les uns des autres qui déclarent chacun des fonctions ayant trait à l'une des fonctionnalités du programme. Par exemple, un fichier pourra contenir les fonctions d'affichage et un autre les fonctions qui font le calcul proprement dit.

Avant d'être exécuté, un fichier source doit être *compilé*, c'est-à-dire traduit en langage machine, langage directement compréhensible par le processeur de l'ordinateur. Un fichier source bien fait peut être *portable*, c'est-à-dire qu'il peut être compilé pour beaucoup de machines différentes, mais, une fois compilé, l'exécutable ne fonctionnera que pour un type de machine déterminé. Par exemple, avec gcc,

```
|| gcc essai.c
```

permet de compiler `essai.c`. Le programme exécutable s'appelle alors, par défaut, `a.out` et se lance en tapant « `./a.out` ». Notez que le fichier exécutable n'a pas besoin de se terminer par « `.exe` » ; le nom est complètement libre. On peut donner un meilleur nom au programme, soit en renommant `a.out` à l'aide de la commande « `mv` », soit en utilisant l'option `-o` de gcc :

```
|| gcc essai.c -o essai
```

En fait, la compilation se fait en plusieurs étapes indépendantes. Le fichier source est d'abord *préprocessé* : les commentaires sont supprimés (et remplacés par un espace), les directives `#include <xxx.h>` sont remplacées par le contenu du fichier `xxx.h`, les macros définies par des `#define` sont remplacées par leur expression, etc. Après cette première étape, le source C est analysé et traduit en assembleur, puis en langage machine. Le résultat, appelé *fichier objet*, contient le code de chacune des fonctions, mais n'est pas encore exécutable. Une dernière étape, appelée *édition de liens*, est nécessaire pour mettre ensemble les différents fichiers objets qui composent le programme, pour prendre en compte les appels faits aux fonctions définies dans des bibliothèques externes, et

pour rajouter un peu d'habillage autour du code du programme pour que le système d'exploitation soit capable de l'exécuter. Il est possible de demander à gcc de s'arrêter après n'importe laquelle de ces étapes.

Un compilateur comme gcc reconnaît de nombreuses options (comme « `-o essai` ») qui peuvent influencer son comportement. Il faut absolument connaître les options suivantes (voir le chapitre 2 pour plus de détails) :

- « `-Wall` » et « `-W` » permettent d'afficher des diagnostics sur le programme. Avec ces options, gcc affiche des avertissements quand il rencontre des constructions suspectes dans le fichier source. Presque toujours, ces constructions suspectes sont des erreurs, et *nous insistons pour que ces deux options soient systématiquement employées*, et pour que le programme soit écrit de telle sorte *qu'aucun avertissement ne soit affiché par gcc*. (Il faut mettre les deux options pour avoir tous les avertissements utiles.)
- « `-lm` » est indispensable pour compiler un programme qui fait appel aux fonctions mathématiques. Cette option doit apparaître après le nom du fichier source.

Plus généralement, « `-lquelquechose` » permet d'utiliser les fonctions définies dans la bibliothèque « `quelquechose` ». La bibliothèque « `m` », qui contient les fonctions mathématiques, est simplement celle la plus couramment utilisée.

Les options `-ansi` (ou `-std=c89`) et `-std=c99` permettent de spécifier dans quelle variante du C le programme est écrit.

L'option `-O` demande au compilateur de passer plus de temps à *la compilation* pour produire un programme qui sera plus efficace à *l'exécution*. Il existe aussi `-O2` et `-O3` sur le même principe, pour des programmes encore plus efficaces.

1.2 Exemple de programme

```
1 | #include <stdio.h>
2 |
3 | void countdown(int count)
4 | {
5 |     while (count>0)
6 |     {
7 |         printf("%d... \n", count);
8 |         count = count - 1;
9 |     }
10 | }
11 |
12 | int main(void)
13 | {
14 |     countdown(6);
15 |     printf("Go !\n");
16 |     return 0;
17 | }
```

Le programme qui précède définit deux fonctions : `countdown()` entre les lignes 3 et 10 et `main()` entre les lignes 12 et 17. En mathématique, habituellement, les fonctions transforment un nombre en un autre nombre ; par exemple : $\sin \frac{\pi}{6}$ donne $\frac{1}{2}$. Ici, « `sin` » est la fonction, et on dira que $\frac{\pi}{6}$ est l'*argument* et $\frac{1}{2}$ est *la valeur de retour* de cette fonction. En C, les fonctions peuvent avoir un nombre arbitraire d'arguments ayant des

types arbitraires (ce ne sont pas forcément des nombres) et ont une ou zéro valeur de retour, de type arbitraire.

Dans l'exemple, la fonction `countdown()` prend un argument, de type entier (`int`), qui est appelé `count` dans la définition de la fonction. Elle n'a pas de valeur de retour (`void`). La fonction `main()` ne prend aucun argument (`void`) et a une valeur de retour de type entier (`int`).

Chaque fonction est composée d'une suite d'instructions écrite entre accolades. Quand une fonction est *exécutée* (c'est-à-dire quand on lui demande de faire son travail), ces instructions sont interprétées dans l'ordre. Quand on compile un programme et qu'on le lance, c'est la fonction `main` qui est exécutée. Ici, la fonction `main()` commence par exécuter la fonction `countdown()` avec l'argument « 6 », c'est-à-dire que les instructions qui composent `countdown()` sont immédiatement interprétées, sachant que l'argument `count` vaut, initialement, 6. Une fois que `countdown()` a fini son travail, la deuxième instruction de la fonction `main()` est exécutée. Il s'agit d'un appel à la fonction `printf()` avec l'argument « "Go!\n" ».

La fonction `printf()` permet d'afficher *une chaîne de caractères* (écrite entre guillemets dans le fichier source) sur l'écran. Cette chaîne de caractère peut contenir des codes spéciaux : ainsi, dans l'exemple, le « `\n` » est remplacé par un retour à la ligne. Dans l'appel à `printf()` de la ligne 7, le « `%d` » présent dans la chaîne de caractères est remplacé par un entier écrit en base 10 (« `d` » pour décimal). L'entier en question est ici le deuxième argument de `printf()`, à savoir la valeur contenue dans la variable `count`. Ainsi, si `count` vaut 6, cet appel à `printf()` affiche « 6. . . » suivi d'un retour à la ligne. Les possibilités de `printf()` seront examinées plus en détail dans le chapitre 6.

La fonction `printf()` n'est pas définie dans le programme ; il s'agit d'une des nombreuses fonctions faisant partie de la bibliothèque standard du C (voir chapitre 7). Ces fonctions ont été définies et compilées une fois pour toutes et peuvent être directement utilisées par tous les programmes C, à condition de déclarer leur existence. C'est le rôle de la ligne « `#include <stdio.h>` » : on dit que le programme inclut l'*entête* `stdio.h`. Presque tous les programmes ont besoin d'utiliser la bibliothèque standard (pour afficher des informations à l'écran) et doivent commencer par cette ligne. D'autres directives `#include` peuvent être nécessaires pour pouvoir utiliser d'autres fonctions provenant de la bibliothèque standard du C ou d'autres bibliothèques. Par exemple, il faut mettre « `#include <math.h>` » au début de tout programme utilisant des fonctions mathématiques (`sin()`, `cos()`, ...). On a également souvent besoin de « `#include <stdlib.h>` » qui définit diverses fonctions utiles, comme `rand()` qui permet d'obtenir des nombres aléatoires. Toutes les lignes commençant par une dièse sont des *directives du préprocesseur* (voir chapitre 8) et sont interprétées lors de la première phase de la compilation.

Après l'appel à `printf()`, la fonction `main()` fait un « `return 0;` ». Ceci n'est pas un appel de fonction : `return` est une directive du C signifiant que la fonction en cours est terminée. Le nombre « 0 » est alors la valeur (de type `int`) que renvoie la fonction `main()`. La fonction `countdown()` n'a pas besoin de directive `return` puisqu'elle ne renvoie pas de valeur (`void`) : l'exécution de `countdown()` s'arrête quand sa dernière instruction a été exécutée. La valeur que renvoie la fonction `main()` est retournée au système d'exploitation. Traditionnellement, cette valeur est un code indiquant si l'exécution du programme s'est bien déroulée.

En plus de `return`, ce petit programme comprend une autre directive du C : il s'agit du `while`. L'instruction se lit naturellement : « TANT QUE (`count` est positif), EXÉCUTER (les lignes entre accolades). Les deux lignes entre accolades après le `while` vont donc être exécutées plusieurs fois, tant que la condition `count > 0` est vraie. Ce genre de construction s'appelle une *boucle*.

Le paramètre `count` de la fonction `countdown()` est une *variable* de *type* `int`. Une variable représente un emplacement dans la mémoire de l'ordinateur, emplacement qui peut contenir des données susceptibles d'évoluer au cours du temps, et le type de la variable (ici, `int`) permet au compilateur de savoir comment interpréter les données dans cet emplacement mémoire. On peut lire une variable (c'est-à-dire lire les données qui sont dans l'emplacement mémoire réservé) ou y écrire. Ainsi, l'instruction « `count = count - 1;` » change la valeur de la variable `count` : le signe « = » est un *opérateur d'affectation* qui permet d'écrire dans l'emplacement mémoire identifié par ce qui se trouve à gauche du signe « = » la valeur de l'*expression* se trouvant à sa droite. Ici, l'ordinateur commence par lire la valeur de `count`, lui retranche 1, puis affecte le résultat de l'opération à la variable `count`. Au cours de l'exécution de `countdown()`, la valeur de la variable `count` va donc diminuer de 1 à chaque exécution de la boucle `while` et passer de 6 (valeur donnée à la variable lors de l'appel depuis `main()`) à 0 (moment où la boucle s'arrête). `count` est une variable un peu particulière, puisque c'est l'un des paramètres de `countdown()`. On peut, bien sûr, déclarer et manipuler des variables qui ne sont pas des paramètres de fonctions.

1.3 Aperçu général de la syntaxe

1.3.1 Commentaires

```
/* La fonction bonjour() permet d'afficher
   le texte « Bonjour ». Cette fonction ne prend
   pas d'argument et n'a pas de valeur de retour */
void bonjour(void)
{
    printf("Bonjour\n"); // Ici, on affiche « bonjour »
    // printf("Hello\n"); // Here, we print « hello »
}
```

Comme dans l'exemple ci-dessus, tout le texte compris entre un « `/*` » et un « `*/` » est ignoré. De même, tout texte compris entre « `//` » et un retour à la ligne est ignoré. De tels commentaires peuvent servir à supprimer effectivement des instructions que l'on ne veut pas complètement effacer (comme, dans l'exemple ci-dessus, la version anglaise de la fonction), mais, surtout, ils servent à décrire et à commenter le code en langage naturel. Il est vivement recommandé de faire de tels commentaires dès qu'une fonction devient un peu compliquée, car cela permet de faciliter grandement les modifications ultérieures d'un programme. (Par contre, la fonction `bonjour()` donnée en exemple est tellement simple qu'on aurait pu ne pas la commenter...)

Les commentaires introduits par « `//` » sont apparus dans la norme C 99, mais étaient déjà reconnus depuis de nombreuses années par la majorité des compilateurs. Il faut bien faire attention au fait que les commentaires « `/* ... */` » ne s'imbriquent pas. Par exemple, dans

```

| | | /* Suppression de la version anglaise
| | | printf("Hello\n"); /* Here, we print hello */
| | | Fin de la partie supprimée */

```

le commentaire commence à « **/* Suppression...** » et se termine au *premier* « ***/** », à savoir juste après « **hello** ». Le compilateur essaye alors d'interpréter la dernière ligne de l'exemple et émet une erreur.

Il est également possible de faire des commentaires avec les directives **#if 0** et **#endif** du préprocesseur. Ce type de commentaires a l'avantage de pouvoir s'imbriquer, contrairement aux « **/*...*/** ».

1.3.2 Identificateurs

Les noms de fonctions (comme **countdown** dans le premier exemple), les noms de variables (comme **count**) et les noms de types (comme **int**) sont des identificateurs. Le choix d'un identificateur est soumis aux règles suivantes :

- Il est exclusivement composé de lettres non accentuées, de chiffres et du caractère « **_** »
- Le premier caractère n'est pas un chiffre
- La *casse* des lettres (majuscule ou minuscule) a une importance : **Valeur** n'est pas le même identificateur que **valeur**.
- Les identificateurs doivent être différents des mots clés du langage. Par exemple, **return** n'est pas un identificateur valide.
- Certains identificateurs sont réservés par la bibliothèque standard. En pratique, il ne faut pas utiliser les noms de fonctions qui y sont définis (comme **printf**, **log**, etc.), et il vaut mieux éviter les identificateurs commençant par un « **_** », « **is** », « **mem** », « **str** », « **to** », « **E** ». (Liste non exhaustive, malheureusement.)

Il est possible que certains compilateurs acceptent des caractères accentués dans les identificateurs, mais il ne faut pas compter dessus. La liste des mots clés du langage est la suivante :

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

En plus des noms de variables, de fonctions et de types, les noms et les membres des **struct**, des **union** et des **enum**, ainsi que les *labels* pour les **goto** et les **case** sont des identificateurs.

1.3.3 Espaces

Presque tous les caractères blancs (c'est-à-dire les espaces, les tabulations et les retours à la ligne) sont inutiles dans un source C. On en a seulement besoin aux endroits où il est évident qu'ils sont nécessaires, à savoir entre deux identificateurs ou mots clés (dans le premier exemple, « **void countdown(int count)** » et non « **voidcountdown(intcount)** »). Cependant, afin d'améliorer la lisibilité des programmes, il est *vivement conseillé* de ne mettre qu'une seule instruction par ligne et d'indenter le programme, c'est-à-dire, comme dans les exemples donnés dans ce document, de mettre en début de ligne un caractère tabulation pour chaque niveau d'imbrication de la ligne dans une paire d'accolades.

Cependant, on ne peut mettre qu'une directive du préprocesseur par ligne.

1.3.4 Fonctions

On définit une fonction en donnant, dans l'ordre, le type de retour de la fonction, le nom de la fonction (un identificateur), une parenthèse ouvrante, la liste des arguments de

la fonction avec leurs types (s'il y a plusieurs arguments, ils sont séparés par des virgules), une parenthèse fermante, une accolade ouvrante, la liste des instructions qui composent la fonction, puis une accolade fermante. Par exemple :

```
|| int marsplus(int x, int y)
|| {
||     return x*y + x + y;
|| }
```

Si la fonction ne prend pas d'argument, la liste des arguments doit être remplacée par « void ». Si la fonction n'a pas de valeur de retour, le type de retour doit être donné par « void ». Si la fonction a une valeur de retour, la fonction doit contenir au moins une directive **return** avec un argument du bon type et tout appel à cette fonction doit atteindre une de ces directives **return**.

Une fonction doit être *déclarée* avant d'être utilisée. *Définir* une fonction est une des manières de déclarer cette fonction, mais il est également possible de déclarer la fonction sans la définir. Pour cela, on donne le type de retour, le nom de la fonction et la liste des types des arguments (avec éventuellement leurs noms), mais, au lieu de donner la liste d'instructions entre accolades, on met un point-virgule. Par exemple, **marsplus** aurait pu être déclarée comme

```
|| int marsplus(int, int);
```

Une fois déclarée ainsi, la fonction peut être utilisée, mais il faudra penser à la définir plus tard. (Sinon, il y aura une erreur à l'édition des liens.) Lorsque déclaration et définition sont faites séparément, il faut bien sûr que le type des arguments et le type de retour soient compatibles.

Une fonction peut s'appeler elle-même ; on dit alors qu'elle est récursive.

1.3.5 Variables de type simple

Les variables représentent des zones de la mémoire de l'ordinateur où l'on peut manipuler des données. Pour déclarer une variable avec un *type simple*, on donne son type puis son nom, suivi d'un point-virgule. (Pour des types plus complexes comme les tableaux, la syntaxe est plus compliquée.) Au moment où elle est déclarée, *la valeur de la variable est complètement arbitraire*. Il est donc nécessaire de donner une valeur à la variable *avant* de lire son contenu. Il est possible d'assigner une valeur à la variable au moment de la déclaration. Par exemple :

```
|| int x;           // Déclaration de la variable x
|| int y=3;        // Déclaration et initialisation de la variable y
|| int z;
||
|| z = x + y;      // INCORRECT; x n'est pas initialisé
```

La dernière ligne de l'exemple est incorrecte, parce qu'on utilise la valeur de **x** sans jamais lui en avoir assigné une. La zone mémoire représentée par **x** contient donc des données *arbitraires* et on ne peut absolument pas prévoir comment le programme va se comporter. La variable **y**, par contre, est correctement utilisée puisqu'elle est initialisée (avec la valeur 3) avant d'être utilisée.

La plupart des exemples de ce chapitre utilisent le type **int** (entier), mais, comme nous le verrons dans le chapitre 3, il existe bien sûr de nombreux autres types dans le langage C. Par exemple, le type **double** permet de représenter des nombres à virgule.

Il est possible de déclarer plusieurs variables d'un coup ; par exemple, on aurait pu remplacer les trois premières lignes de l'exemple précédent par « `int x, y=3, z;` ». La déclaration d'une variable fait deux choses : elle réserve un espace dans la mémoire de l'ordinateur et associe un identificateur (le nom de la variable) à cet espace. Si l'on veut avoir un programme fait de plusieurs fichiers avec une variable commune à deux fichiers, il est possible de déclarer cette variable normalement dans un des fichiers (et donc de lui allouer un espace dans la mémoire) et de déclarer cette variable dans l'autre fichier avec le mot clé `extern`, qui indique au compilateur de ne pas réserver d'espace dans la mémoire de l'ordinateur.

1.3.6 Variables de type complexe : tableaux

Un tableau est une agrégation de plusieurs variables de même type. Ainsi

```
|| int table [ 5 ];
```

déclare un tableau nommé `table` ayant cinq éléments de type `int`. Tout se passe comme si ces cinq éléments, nommés `table[0]`, `table[1]`, `table[2]`, `table[3]` et `table[4]`, étaient cinq variables indépendantes de type `int`. Attention, l'index (c'est à dire le numéro) qui repère les éléments d'un tableau de taille n varie entre 0 et $n - 1$; ainsi, dans l'exemple précédent, `table[5]` *n'est pas un élément valide !* L'avantage d'un tableau est que l'index d'un élément peut être une expression arbitrairement compliquée. Ainsi, par exemple, si `idx` est une variable type entier, `table[idx-1]` dénote l'élément du tableau dont l'index est le résultat de l'opération « `idx-1` ». (Et il *faut* que la valeur de `idx` soit comprise entre 1 et 5, sinon l'index ne serait pas compris entre 0 et 4 et le programme ne serait pas correct.)

1.3.7 Variables globales et locales

Une variable peut être déclarée dans le fichier source à l'extérieur de toute fonction. Cette variable peut alors être utilisée par toutes les fonctions du programme qui suivent la déclaration. On dit alors que la variable est *globale*. Une variable peut également être déclarée à l'intérieur d'une fonction. Dans ce cas, cette variable ne peut être utilisée par les instructions qui suivent sa déclaration que dans le bloc (délimité par des accolades) où elle a été définie. Une telle variable est dite *locale*. Par exemple

```
|| int dots = 5; // variable globale ; on peut l'utiliser partout  
|| void countdown(int n) // le paramètre n est une variable locale  
|| {  
||     printf("Compte à rebours\n");  
||     while ( n > 0 )  
||     {  
||         printf("%d", n);  
||         int i = dots; // i est une variable locale  
||         while ( i > 0 )  
||         {  
||             printf(".");  
||             i = i - 1;  
||         }  
||         n = n - 1;  
||     }  
||     printf("Go !\n"); // au delà de ce point, i n'est plus défini  
|| } // au delà de ce point, n n'est plus défini
```

Le nom d'une variable locale ne peut pas être utilisé pour accéder à cette variable en dehors du bloc où elle a été déclarée. Dans l'exemple ci-dessus, le nom de variable `n` n'est défini *que* dans la fonction `countdown()` et ne peut donc pas être utilisé ailleurs. De même, le nom de variable `i` ne peut être utilisé que dans la boucle du premier `while`.

Si `countdown()` appelait une autre fonction en lui passant un pointeur sur `n`, cette autre fonction pourrait modifier la valeur de `n` en utilisant ce pointeur, mais pas en utilisant le nom `n`. On dit que `n` est un identificateur à portée locale.

Lorsque l'exécution d'un bloc d'instructions donné se termine, toutes les variables locales qui y sont déclarées sont « détruites » : l'espace dans la mémoire qu'elles utilisaient ne leur est plus réservé et les données qu'elles contenaient sont perdues. En particulier, lors d'une exécution ultérieure du même bloc, une variable locale aura oublié sa valeur précédente.

Il est donc illégal de prendre un pointeur sur une variable locale et de déréférencer ce pointeur une fois terminée l'exécution du bloc contenant cette variable locale.

Cependant, si une variable locale est déclarée avec le mot clé `static`, elle a un espace mémoire stable qui lui est réservé une bonne fois pour toute, la variable conservera sa valeur entre deux exécutions du même bloc, et on peut garder des pointeurs sur cette variable.

En général, et contrairement à l'exemple donné ci-dessus, on déclare les variables locales au tout début d'un bloc, et même, le plus souvent, au tout début d'une fonction. Cette pratique était d'ailleurs obligatoire avant C 99.

Une variable locale peut avoir le même nom qu'une variable globale ou qu'une autre variable locale déclarée dans un bloc englobant le bloc courant. Dans ce cas, seule la variable déclarée dans le bloc le plus profond qui soit encore ouvert peut être accédée par ce nom. Par exemple

```
|| int i = 4;
|| {
||     int i = 3;
||     printf("%d\n",i);           // affiche 3
|| }
|| printf("%d\n",i);           // affiche 4
```

Cette pratique est vivement déconseillée.

1.3.8 Instructions simples, boucles et tests

Une instruction simple peut être un appel de fonction, comme « `printf("Hi !\n");` », une affectation à une variable, comme « `count = count - 1;` », ou un appel à une directive du C, comme « `return 0;` ». Dans tous les cas, une instruction simple se termine par un point-virgule.

Le C dispose de plusieurs constructions plus complexes permettant de faire des *tests* ou des *boucles*. On a déjà vu une boucle dans le premier exemple de ce chapitre : `while (test) {liste d'instructions}`. Voici un exemple de test :

```
|| printf("Il y a %d résultat", results); // On n'affiche pas de retour à la ligne !
|| if (results > 1)
|| {
||     printf("s");           // Marque du pluriel
|| }
|| printf("\n");           // On affiche juste le retour à la ligne
```

La syntaxe est très similaire au `while` : SI la condition entre parenthèses `results > 1` est vraie, ALORS les instructions entre accolades sont exécutées (juste une fois, contrairement

au `while`). Notez que l'on a indenté les instructions entre accolades pour bien marquer visuellement le fait qu'elles étaient spéciales. On peut rajouter un bloc `else` à une telle construction, qui n'est exécuté que si le résultat du test est faux. Par exemple :

```

| printf("Il y a %d cheva", results);
| if (results > 1)
| {
|     printf("ux\n");
| } else
| {
|     printf("l\n");
| }

```

En fait, un bloc d'instructions écrites à l'intérieur d'une paire d'accolades est, à tout point de vue, considéré par le C comme une instruction unique. De plus, il n'est pas obligatoire d'utiliser de telles instructions composées dans des constructions de tests ou de boucles; on peut utiliser des instructions simples. Ainsi, l'exemple plus haut aurait pu être écrit

```

| printf("Il y a %d cheva", results);
| if (results > 1) printf("ux\n"); else printf("l\n");

```

(notez la position des points-virgules). Sauf peut-être dans les cas les plus simples, il vaut mieux toujours s'astreindre, dans un premier temps, à rajouter une ou deux paires d'accolades dans de telles constructions.

En C, la « condition entre parenthèses » qui apparaît dans une boucle ou dans un test peut être une expression très compliquée mais, dans un premier temps, il vaut mieux se contenter d'utiliser les opérateurs `<`, `<=`, `>`, `>=`, `==` (est égal à) et `!=` (est différent de). Attention au piège : « `==` » signifie « est égal à » et sert dans les tests et les boucles, alors que « `=` » signifie que l'on donne à ce qui est à gauche la valeur de ce qui est à droite :

```

| int n = 4; // Un seul signe égal; on donne la valeur 4 à la variable n
| if ( 2 + 2 == 4 ) // Deux signes égal; on teste l'égalité de 2+2 et de 4
| {
|     ... // Ceci sera exécuté, puisque le test est vrai
| }

```

Se méfier...

Comme nous le verrons en détail dans le chapitre 5, en plus de la boucle `while`, le C dispose de la boucle `for` et de la boucle `do`. En plus du test `if`, le C dispose du test `switch`. Mentionnons simplement ici l'usage le plus courant de la boucle `for` :

```

| int n;
| for (n=0 ; n<20 ; n++)
| {
|     ...
| }

```

Dans cet exemple, la liste d'instructions entre les accolades sera, sauf cas spécial, exécutée 20 fois. La première fois, `n` vaut 0, la deuxième fois `n` vaut 1, etc., et la dernière fois `n` vaut 19. Le « `n=0` » est l'*initialisation* de la boucle. Le « `n<20` » indique à quelle *condition* la boucle doit continuer à tourner et le « `n++` » signifie que la variable « `n` » doit être augmentée de 1 après chaque tour de la boucle. Une manière équivalente d'écrire le même fragment est donc :

```

int n=0;
while (n < 20)
{
    ...
    n = n + 1;
}

```

(En fait, on aurait pu écrire `n=n+1` » au lieu de « `n++` » dans la boucle « `for` ».)

1.3.9 Expressions et opérateurs

Dans le premier exemple de ce chapitre, l'instruction « `count = count - 1;` » assigne à la variable `count`, à gauche du signe égal, la valeur de l'expression « `count - 1` », à droite du signe égal. Cette expression à droite du signe égal est d'abord évaluée (l'ordinateur lit la valeur actuelle de `count` et retranche 1) avant d'être assignée à `count`. Insistons sur ce point : une affectation avec un signe égal est effectuée immédiatement, avec la valeur actuelle de l'expression à droite du signe égal : en C, « `a=x;` » signifie que `a` prend *tout de suite* la valeur que `x` a à ce moment. Ça ne signifie pas que `a` aura toujours la même valeur que `x`, ce n'est pas une égalité mathématique.

Dans l'appel de fonction « `countdown(6);` », le nombre 6 est également une expression, même si c'est l'expression la plus simple que l'on puisse imaginer. On aurait obtenu le même résultat en écrivant « `countdown(3*5-9);` ». Dans ce dernier cas, l'ordinateur aurait d'abord calculé l'expression « `3*5-9` » puis appelé la fonction `countdown()` avec le résultat de l'évaluation, à savoir 6. La fonction `countdown()` n'a aucun moyen de savoir si la valeur qu'elle reçoit provient d'une expression constante, d'une variable ou d'un calcul compliqué.

Une expression peut être une valeur constante (comme le nombre 6, par exemple), ou la valeur contenue dans une variable, ou une combinaison de plusieurs expressions plus simples à l'aide d'un *opérateur*. Par exemple, le signe moins représente l'opérateur soustraction.

Le C a un grand nombre d'opérateurs (voir chapitre 4) ; en les combinant entre eux, on peut obtenir des expressions arbitrairement compliquées. Une des difficultés du C est de savoir *dans quel ordre* les différents opérateurs sont appliqués : il y a des règles, pas toujours intuitives, indiquant les priorités relatives des différents opérateurs. Par exemple, l'opérateur multiplication (*) est prioritaire devant l'opérateur addition (+), et l'expression « `2 + 3 * 4` » vaut bien 14 et non pas 20. Il est toujours possible d'imposer l'ordre de l'évaluation avec une paire de parenthèses. Par exemple, « `(2 + 3) * 4` » est une expression valide qui vaut 20.

Lorsqu'on a des doutes sur les priorités relatives des opérateurs, il n'y a aucune honte à mettre une paire de parenthèses en trop, pour être sûr que le calcul se fait bien dans le bon sens. Par exemple, il vaut mieux écrire « `(10 / 5) * 2` » plutôt que « `10 / 5 * 2` ». Ces deux expressions valent 4, mais la première est plus claire et assure que personne ne pensera (à tort) que l'expression vaut « `10 / (5 * 2)` ».

Le signe égal, qui marque l'affectation à une variable, est un opérateur comme un autre, avec une très basse priorité. En particulier, on peut changer l'ordre d'évaluation avec des parenthèses et l'expression « `(x = 3 + y) / 2` » est valide et a un sens différent de l'expression « `x = 3 + y / 2` ». Dans ce cours, nous ne nous aventurerons pas dans ce genre de style de programmation.

Certains opérateurs du C se ressemblent beaucoup mais ont un sens très différent. Ainsi, encore une fois, il ne faut pas confondre l'opérateur d'affectation = avec l'opérateur de test ==.

1.4 Directives du préprocesseur

Les directives du préprocesseur sont les lignes commençant par # dans le code source. Ces lignes ont une syntaxe différente des autres et sont interprétées avant la compilation proprement dite du programme. Deux directives du préprocesseur doivent être connues : `#include`, qui permet de charger des définitions de fonctions appartenant, par exemple, à la bibliothèque standard du C (`#include <stdio.h>` pour l'affichage, `#include <math.h>` pour les fonctions mathématiques, etc.), et `#define` qui permet de déclarer des expressions constantes. Par exemple, écrire

```
|| #define N 300
```

au début du programme fera que l'identificateur N sera remplacé par le nombre 300 partout dans le programme. On dit que N est une « macro du préprocesseur » ou, en plus court, une macro. Les macros ne sont pas des variables, on n'a pas le droit d'écrire « N=500; » dans le programme. En pratique, tout se passe comme si on modifiait le programme et qu'on écrivait 300 partout où apparaît l'identificateur N.

Les macros permettent d'écrire *en un seul endroit* la valeur d'un paramètre que l'on veut utiliser plusieurs fois dans le programme. Par exemple, 300 pourrait être la taille du système que l'on veut étudier avec l'ordinateur. Il est probable qu'on ait besoin de ce nombre en plusieurs endroits dans le programme et, si on a utilisé une macro, il est très facile de changer sa valeur pour étudier un système de taille différente.

Contrairement aux instructions, les directives du préprocesseur ne prennent pas de point-virgule final.

Chapitre 2

Écriture du programme et compilation

2.1 Savoir corriger les erreurs

2.1.1 Corriger un programme pour qu'il compile

Un compilateur C est extrêmement exigeant sur la syntaxe du fichier source. Il suffit d'un point-virgule manquant, ou d'une paire d'accolades mal refermée pour que le programme ne soit plus *valide* et que le compilateur refuse de compiler.

Si tel est le cas, le compilateur affichera un ou plusieurs messages d'erreur. Afin de pouvoir corriger les programmes que l'on est en train d'écrire, il est important de lire et de comprendre ces messages d'erreur. Ces messages peuvent varier d'un compilateur à l'autre ; dans ce cours, nous utilisons gcc.

Prenons un exemple de programme avec plusieurs erreurs ; supposons que le fichier `essai.c` contienne

```
1 | #include <stdio.h>
2 |
3 | // Ce programme contient plusieurs erreurs
4 |
5 | int main(void)
6 | {
7 |     int a = 3
8 |     printf("L'exponentielle de %d vaut %d\n", a, exp(a) );
9 |     return 0;
10| }
```

Si on essaye de compiler avec

```
|| gcc essai.c
```

le compilateur refuse de compiler et affiche

```
|| essai.c: In function 'main':
|| essai.c:8: error: syntax error before "printf"
|| essai.c:8:16: missing terminating " character
```

(Ces messages peuvent éventuellement varier légèrement d'une version de gcc à l'autre. Ils peuvent aussi apparaître dans une langue qui est un étrange mélange entre l'anglais et

le français.)

Pour faciliter le travail du programmeur, le compilateur essaye de localiser le plus précisément possible l'endroit où l'erreur a été commise :

- Au début de chaque ligne d'erreur, le nom du fichier source (ici, `essai.c`) est affiché.
- Avant d'afficher un message d'erreur, le compilateur indique dans quelle fonction il se trouve. Ici, la première ligne du message nous apprend que les deux erreurs que gcc a trouvées sont dans la fonction `main()`.
- Pour chacune des erreurs reportées, le compilateur indique un *numéro de ligne* (ici, `8`, dans les deux cas) et, éventuellement, une position dans la ligne (ici, pour la deuxième erreur, le compilateur suggère de s'intéresser au seizième caractère de la ligne).

Le reste du texte explicite l'erreur. Une « `syntax error` » est probablement l'erreur la plus courante. Elle est le plus souvent due à un oubli dans la ponctuation du source. Ici, il manque le point-virgule après la déclaration et l'initialisation de la variable `n` : il fallait écrire « `int n = 3;` ». Notez que l'erreur se trouve à la ligne 7 et non à la ligne 8 ; *les numéros de ligne reportés par le compilateur ne sont pas parfaitement fiables*, et il faut toujours regarder si l'erreur ne se trouve pas quelques lignes au dessus de l'endroit suggéré.

La deuxième erreur indique qu'il manque un guillemet double (« " ») pour terminer la chaîne de caractères affichée par `printf()`. Ici, le numéro de ligne est bon, et la position dans la ligne indique le « " » qui a commencé la chaîne de caractères. Pour corriger cette erreur, il suffit donc de rajouter un guillemet double juste avant la virgule.

Une fois ces deux erreurs corrigées, on peut essayer de relancer la compilation :

```
|| gcc essai.c
```

gcc affiche une nouvelle erreur, d'un type différent :

```
|| /tmp/ccAkgcZ2.o(.text+0x28): In function 'main':  
|| : undefined reference to 'exp'  
|| collect2: ld returned 1 exit status
```

en fait, cette erreur n'a pas été produite pendant la compilation, mais pendant l'édition de liens. C'est pour cette raison qu'elle a un aspect différent.

Ici, la partie importante est « `undefined reference to 'exp'` ». La fonction `exp` n'est disponible que dans la bibliothèque mathématique. Pour obtenir un exécutable, il faut préciser à gcc que l'on utilise des fonctions de cette bibliothèque et taper

```
|| gcc essai.c -lm
```

Maintenant, le programme compile et un exécutable est produit. Ça ne veut pas dire que le programme est correct et fonctionne comme prévu...

2.1.2 Corriger un programme pour qu'il marche

Si on essaye d'exécuter le programme compilé dans la section précédente, on obtient

```
|| L'exponentielle de 3 vaut -1083199226
```

Ce résultat est certainement faux ! Notre programme est toujours incorrect. On s'en serait rendu compte si l'on avait demandé à gcc de faire des avertissements avec « `-Wall` » et « `-W` » :

```
|| gcc -Wall -W essai.c -lm
```

On obtient :

```
|| essai.c: In function 'main':  
|| essai.c:8: warning: implicit declaration of function 'exp'  
|| essai.c:8: warning: int format, double arg (arg 3)
```

Le mot « **warning** » est affiché pour indiquer que l'on a des avertissements et non des erreurs, mais, à part ça, le message ressemble à ce qu'on avait plus haut. Le premier avertissement fait remarquer que l'on utilise la fonction **exp** sans jamais l'avoir déclaré. En effet, comme pour toutes les fonctions mathématiques, il faut penser à rajouter « **#include <math.h>** » au début du fichier pour avoir le droit d'utiliser **exp**. Insistons sur ce point important : pour pouvoir utiliser **exp**, il faut mettre « **#include <math.h>** » dans le fichier *et* compiler avec l'option **-lm**.

le **#include** est nécessaire pour la compilation, et le **-lm** est nécessaire pour l'édition de liens.

L'autre problème que reporte **gcc** est que le troisième argument de **printf** (à savoir, « **exp(a)** ») est de type **double** (c'est-à-dire que **exp** renvoie un nombre à virgule et non un entier), alors que le format « **%d** » dans la chaîne de caractère indiquait que l'on attendait un **int**. En effet, comme nous le verrons dans le chapitre 6, « **%d** » permet seulement d'afficher des entiers et il faut utiliser le format « **%g** » pour afficher des **double**.

Au final, le programme corrigé s'écrit donc

```
|| #include <stdio.h>  
|| #include <math.h>  
  
|| int main(void) {  
||     int a = 3;  
||     printf("L'exponentielle de %d vaut %g\n", a, exp(a) );  
||     return 0;  
|| }
```

Ce programme compile sans erreurs ni avertissements et fonctionne correctement.

Il est *indispensable* de *systématiquement* utiliser les options « **-Wall** » et « **-W** » de **gcc** et de tenir compte de tous les avertissements qu'il pourrait émettre. Dans presque tous les cas, chacun de ces avertissements est une erreur qui va faire que le programme ne se comportera pas comme prévu.

2.1.3 Quelques conseils

- Toujours utiliser les options « **-Wall** » et « **-W** » de **gcc**, et prendre les avertissements au sérieux. (On ne le répétera jamais assez.)
- Si **gcc** affiche des centaines de lignes d'erreur, ça ne veut pas dire qu'il y a des centaines d'erreurs dans le programme. Il peut arriver que l'oubli d'un ou deux caractères dans le source engendre beaucoup de lignes d'erreur. Commencez par corriger les premières erreurs affichées et recompilez pour voir si, par hasard, les autres erreurs n'ont pas disparu.
- Si le programme compile sans erreurs ni avertissements mais ne fonctionne pas comme prévu, c'est qu'il y a une erreur dans la logique du programme ; ce sont les

problèmes les plus difficiles à corriger, et il n'y a pas de recette magique. Une bonne méthode, qui marche souvent, consiste à rajouter temporairement des lignes comme « `printf("La variable x vaut %d\n", x);` » pour vérifier, au fur et à mesure de l'exécution, que tout se déroule comme prévu.

- N'oubliez pas de mettre des retours à la ligne (`\n`) à la fin des chaînes de caractères que vous affichez, sinon vous risquez de ne pas les voir.
- À partir du moment où vous utilisez des tableaux ou des chaînes de caractères, d'autres difficultés apparaissent. Dans un tableau de n éléments, l'index varie de 0 à $n - 1$. Si on essaye de lire ou d'écrire un élément du tableau avec un index plus petit que 0 ou plus grand que $n - 1$, *il n'y aura pas d'erreur à la compilation* et le programme fera *n'importe quoi*. Le programmeur travaille sans filet et doit s'assurer que la logique de son programme est correcte et qu'il n'y aura pas d'accès en dehors des limites du tableau.
- Pour des programmes encore plus compliqués qui interagissent avec leur environnement, d'autres pièges apparaissent. Comment se comportera votre programme s'il essaye d'écrire dans un fichier et qu'il n'y a plus de place sur le disque dur ? Ou s'il demande à l'utilisateur de choisir un nombre entre 1 et 10 et que l'utilisateur tape une série de 1 000 caractères aléatoires ? Un programme bien écrit doit se comporter intelligemment dans toutes les situations et un programmeur chevronné doit prendre en compte tous les événements possibles, mêmes improbables, qui pourraient influencer le déroulement du programme. C'est ce manque d'attention envers les événements inhabituels qui fait que la grande majorité des programmes existants ont des « trous de sécurité » qui permettent aux virus informatiques de prendre le contrôle d'ordinateurs et de se propager.

2.2 Comment écrire un programme

Expliquer comment un programme doit être écrit est difficile : toutes les situations sont différentes, et il faut savoir s'adapter au problème à résoudre. Tout cela demande surtout de la pratique.

On peut néanmoins énoncer un grand principe général :

Le programme doit être agréable à lire et facilement compréhensible.

Quand on cherche une erreur, qu'on étudie un programme ou qu'on veut en modifier, il faut qu'on puisse se repérer facilement et comprendre rapidement n'importe quelle portion du code source. Il faut donc

- Indenter le programme, pour que sa structure logique saute aux yeux : il faut mettre une instruction par ligne, et commencer chaque ligne par autant de caractères tabulation qu'il y a de niveaux d'imbrication dans des accolades.
- Utiliser des noms de variables et de fonctions explicites (par exemple `x_moyen`, ou `affiche_table()`).
- Mettre des commentaires pour décrire, dès que ce n'est plus évident, ce que fait une fonction, ou la quantité que contient une variable.
- Éviter les fonctions trop grandes et trop complexes qui font plein de choses, et leur préférer plusieurs fonctions plus petites spécialisées dans une seule tâche.

- En particulier, il faut éviter de tout mettre dans la fonction `main()`. Souvent, la fonction `main()` peut se réduire à quelque chose qui ressemble à

```

int main(void) {
    int t = 0;
    initialiser_le_systeme ();
    while (t < T_MAX) {
        faire_une_etape();
        afficher_le_systeme();
        t = t + 1;
    }
    printf("À la fin, le résultat est %d\n", calculer_le_resultat() );

    return 0;
}

```

(Dans cet exemple, on a supposé qu'il y avait un tableau en variable globale qui contenait l'état du système. Les deux fonctions `initialiser_le_systeme()` et `faire_une_etape()` modifient ce tableau et les fonctions `afficher_le_resultat()` et `calculer_le_resultat()` lisent les valeurs du tableau.)

- Il ne doit pas y avoir de surprise dans les fonctions : si, dans l'exemple ci-dessus, la fonction `afficher_le_systeme()` modifie l'état du système, c'est la catastrophe ! Personne ne peut s'attendre à ce que la fonction `afficher_le_systeme()` fasse autre chose que d'afficher le système. Si c'est le cas, il est certain qu'il y aura un jour un bug dans ce programme parce que le programmeur aura oublié cette invraisemblable particularité.
- Pour les mêmes raisons, il vaut mieux éviter autant que possible les variables globales et utiliser les arguments et les valeurs de retour des fonctions pour faire circuler l'information. Le problème des variables globales est qu'on risque, très facilement, d'oublier quelles fonctions modifient quelles variables et donc d'être surpris par le comportement d'une fonction. Idéalement, quand une ligne modifie la valeur d'une variable, la variable est définie moins de trente lignes plus haut. On peut admettre d'utiliser une variable globale pour contenir une quantité qui ne sera jamais modifiée mais qui est lue par presque toutes les fonctions (par exemple, le programme demande à l'utilisateur d'entrer la valeur de la taille du système au tout début du programme et écrit cette valeur dans la variable globale `taille`. Cette valeur n'est plus jamais modifiée par la suite). Quand on débute en C, on peut également admettre de mettre les tableaux en variables globales (passer des tableaux en argument à une fonction est possible, mais plein de subtilités) et, exceptionnellement, utiliser une variable globale quand on aurait besoin qu'une fonction renvoie *deux* valeurs différentes (on peut faire autrement, mais c'est un peu compliqué). *Tout autre emploi des variables globales est à bannir.*
- Éviter d'utiliser des constantes numériques dans le programme. Les tailles de tableaux, les constantes physiques, le nombre d'itérations dans une boucle ne doivent pas apparaître sous forme numérique dans le programme mais sous forme symbolique, à l'aide d'une macro (déclarée avec `#define`) ou d'une variable.

2.3 Un exemple complet

De combien de manières différentes peut-on écrire le nombre 38 464 comme la somme de trois carrés parfaits ?

Autrement dit, combien y a-t-il de triplets de nombre naturels (n_1, n_2, n_3) tels que $n_1^2 + n_2^2 + n_3^2 = 38\,464$? Il serait absurde de chercher la solution à la main, et il ne semble pas y avoir de manière astucieuse de répondre à la question. Utiliser un ordinateur est la seule méthode raisonnable.

Comment s'y prendre ? En testant tous les triplets possibles un par un. À chaque fois qu'un triplet convient, on augmente de 1 la valeur d'une certaine variable qui nous servira de compteur. À la fin, il ne reste plus qu'à afficher ce compteur

```
|| Pour tous les triplets (n1, n2, n3) {  
||     Si le triplet convient {  
||         compteur = compteur + 1;  
||     }  
|| }  
|| Afficher la valeur du compteur
```

Le compteur sera une variable de type `int`, qu'il ne faut pas oublier d'initialiser à 0 (si aucun triplet convient, c'est la bonne réponse).

Nous allons écrire plusieurs versions de ce programme. Chaque version sera meilleure que la précédente.

2.3.1 Un *hack*

La version proposée dans cette section fonctionne, mais n'est pas vraiment acceptable.

```
1 | #include <stdio.h>  
2 |  
3 | int main(void) {  
4 |     int n1,n2,n3;  
5 |     int compteur=0;  
6 |  
7 |     for (n1=0; n1<200; n1++) {  
8 |         for (n2=0; n2<200; n2++) {  
9 |             for (n3=0; n3<200; n3++) {  
10 |                 if (n1*n1+n2*n2+n3*n3 == 38464) {  
11 |                     printf("%d %d %d\n",n1,n2,n3);  
12 |                     compteur = compteur + 1;  
13 |                 }  
14 |             }  
15 |         }  
16 |     }  
17 |     printf("Réponse: %d.\n",compteur);  
18 |     return 0;  
19 | }
```

Comme on utilise `printf`, On a écrit, en haut du programme, `#include <stdio.h>` pour déclarer les fonctions ayant trait à l'affichage.

On emploie ici des boucles `for` (voir page 12), mais on aurait aussi bien pu employer des boucles `while`. Ce qui est sûr, c'est que si l'on veut faire quelque chose avec tous les triplets de nombres (n_1, n_2, n_3) (ici, vérifier si la somme de leurs carrés vaut ou non 38 464), il faut faire *trois boucles imbriquées les unes dans les autres*. La première boucle `for` fait varier n_1 de 0 à 199, et pour chacune de ces valeurs de n_1 , on a une deuxième boucle `for` qui fait varier n_2 de 0 à 199, et pour chacune de ces valeurs de n_1 et n_2 , la troisième boucle `for` fait varier n_3 de 0 à 199. À l'intérieur de ces trois boucles `for` emboîtées les unes dans les autres se trouve le test `if` qui vérifie si le triplet convient ou non. Ce test `if` sera exécuté $200^3 = 8\,10^6$ avec, dans cet ordre, (n_1, n_2, n_3) qui valent $(0,0,0)$, $(0,0,1)$, $\dots$, $(0,0,199)$, $(0,1,0)$, $\dots$, $(0,1,199)$, $(0,2,0)$, $\dots$, $(0,199,199)$, $(1,0,0)$, $\dots$, $(199,199,199)$.

A priori, on veut tester tous les triplets d'entiers naturels (n_1, n_2, n_3) . Pourquoi, alors, limite-t-on chacun de ces nombres à 199? D'abord, il y a un nombre infini de triplets d'entiers naturels et il est impossible, avec un ordinateur, de faire un nombre infini d'opérations. Ensuite, il n'est pas difficile de voir que si n_1 (ou n_2 , ou n_3) est supérieur ou égal à 200, alors n_1^2 est déjà strictement plus grand que 38 464 et il n'y aura aucune solution.

Si on tape ce programme dans un éditeur de texte, qu'on le sauve dans le fichier `essai1.c`, qu'on le compile avec `gcc -Wall -Wessai1.c -o essai1` et, enfin, qu'on l'exécute avec `./essai1`, on obtient toutes les solutions.

```

| 0 40 192
| 0 192 40
| 24 32 192
| [... 25 lignes ont été supprimées]
| 184 48 48
| 192 0 40
| 192 24 32
| 192 32 24
| 192 40 0
| Réponse: 33

```

On a affiché non seulement la réponse qui nous intéresse (33 triplets) mais aussi la liste des triplets, ce qui n'était pas demandée.

2.3.2 Écrire un programme plus général

Un des problèmes du programme de la section précédente est qu'il ne fonctionne *que* pour le nombre 38 464. Si on veut le nombre de triplets dont la somme des carrés vaut 48 854, il faut modifier, bien sûr, la ligne 10 (remplacer 38 464 par 48 854), mais aussi les lignes 7, 8 et 9 pour remplacer 200 par quelque chose de plus grand ($48\,854 = 210^2 + 23^2 + 65^2$). Ça fait quatre modifications, et le risque est trop grand qu'on en oublie une et que l'on obtienne un programme incorrect. Si on imagine qu'on regarde ce programme dans un an ou deux, on risque fort d'avoir oublié le raisonnement qui a conduit à mettre ce « 200 » aux lignes 7, 8 et 9, et on risque d'avoir un problème pour déterminer par quoi il faut le remplacer. De manière générale, il faut éviter d'avoir des constantes numériques « magiques » dans les programmes, qui ont été difficiles à déterminer et qu'il sera difficile

de modifier. Il vaut bien mieux faire calculer par le programme ce nombre « 200 » qui apparaît dans les trois boucles.

Un autre problème est que tout le programme est dans la fonction « `main()` », ce qui rend le programme peu réutilisable. Voici donc une meilleure version de ce programme.

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

int troiscarres (int objectif) {
    int nmax = sqrt(objectif);

    int n1, n2, n3;
    int compteur = 0;

    for (n1=0; n1<=nmax; n1++) {
        for (n2=0; n2<=nmax; n2++) {
            for (n3=0; n3<=nmax; n3++) {
                if (n1*n1+n2*n2+n3*n3 == objectif) {
                    compteur = compteur + 1;
                }
            }
        }
    }

    return compteur;
}

int main(void) {
    printf("Réponse: %d.\n", troiscarres(38464) );
    return 0;
}
```

Cette nouvelle version utilise la fonction `sqrt()` qui calcule la racine carrée d'un nombre (« **square root** » en anglais). `sqrt()` est définie dans la bibliothèque de fonctions mathématiques du C et, pour s'en servir, il faut rajouter `#include <math.h>` en haut du programme et compiler avec l'option `-lm` de `gcc`. Par exemple, si le source du programme est dans le fichier `essai2.c`, on tape « `gcc -Wall -W essai2.c -lm -o essai2` ».

Cette fonction est déclarée comme `double sqrt(double)`, ce qui signifie qu'elle prend un argument de type `double` et renvoie un nombre de type `double`, c'est-à-dire des nombres à virgule. Ici, on l'utilise avec un argument de type entier (`int`) et on écrit le résultat dans une variable de type entier. Le C va tout seul convertir l'argument de `sqrt()` en double, appeler `sqrt()`, puis convertir le résultat en `int` avant de l'écrire dans la variable `nmax`.

Il y a maintenant deux fonctions dans le programme. L'une, `troiscarres()`, fait le travail, et l'autre, la fonction `main()`, appelle `troiscarres()` et affiche le résultat. La fonction `troiscarres()` prend un argument de type entier qui est le nombre qu'il faut décomposer, et retourne à l'appelant (ici, `main()`) la réponse souhaitée. Il faut noter que ce n'est pas `troiscarres()` mais `main()` qui affiche le résultat. Ainsi, on a bien séparé la fonction qui fait le travail de celle qui l'exploite (ici, en affichant le résultat), et on pourra réutiliser telle quel la fonction `troiscarres()` dans un autre programme.

Il n'y a plus aucun nombre « magique » dans la fonction `troiscarres()`. L'objectif

(38 464) apparaît seulement dans la fonction `main()`, et on fait varier `n1`, `n2` et `n3` de 0 à `nmax` inclus, où `nmax` est la racine carré de l'objectif. (Après tout, si $n_1 > \sqrt{X}$, on est sûr que $n_1^2 + n_2^2 + n_3^2 \neq X$.) Notez que le test dans les boucles `for` est `<= nmax` et non `< nmax`. En effet, si l'objectif est un carré parfait, alors le triplet d'entier $(\sqrt{\text{objectif}}, 0, 0)$ convient et il faut bien que `n1` aille jusqu'à `nmax` *inclus*.

Si on veut le nombre de triplets pour 48 854, il y a seulement *un* changement à faire dans le programme. Et si on veut que le programme demande un nombre à l'utilisateur et affiche le nombre de triplets correspondants, il suffira de modifier la fonction `main()` sans rien toucher à `troiscarres()`.

2.3.3 Performance

Le programme de la section précédente est acceptable, mais pas très performant. C'est toujours une bonne chose que d'essayer d'estimer le nombre d'opérations que va devoir exécuter un programme. Si on veut le nombre de triplets dont la somme des carrés vaut X , combien d'opération va-t-on exécuter ? Ici, c'est facile. Le gros du programme est constitué de trois boucles imbriquées, chaque boucle va de 0 à $\sqrt{X}$ donc le `if` à l'intérieur des trois boucles sera exécuté $(\sqrt{X})^3 = X^{3/2}$ fois. Ça signifie que si on donne au programme un nombre de l'ordre de $4 \cdot 10^6$ (c'est-à-dire 100 fois plus grand que 38 464), le programme mettra $100^{3/2} = 1\,000$ fois plus de temps à s'exécuter. C'est beaucoup.

Dans ce cas particulier, on peut aisément optimiser le programme. En effet, la dernière boucle (sur `n3`) est inutile. Une fois que l'on a choisi `n1` et `n2`, on a juste besoin de vérifier que le nombre qui reste à ajouter est un carré parfait.

```
int troiscarres(int objectif) {
    int nmax = sqrt(objectif);

    int n1, n2;
    int compteur = 0;

    for (n1=0; n1<=nmax; n1++) {
        for (n2=0; n2<=nmax; n2++) {
            int reste = objectif - n1*n1 - n2*n2;
            if (reste >=0) {
                int n3 = sqrt(reste);
                if (n3*n3 == reste) {
                    compteur = compteur + 1;
                }
            }
        }
    }

    return compteur;
}
```

Notez qu'on a juste remis la fonction `troiscarres` ; le reste du programme n'a pas changé. Dans cette version, on fait deux boucles : une sur `n1` et une sur `n2`, et on calcule le nombre `reste`. Si ce nombre est un carré parfait, on a une solution et on incrémente la variable

`compteur`. On commence par vérifier que `reste` est positif puis on met dans l'entier `n3` la racine carré de `reste`. A priori, la racine carré de `reste` n'est pas un entier, donc on perd tous les chiffres après la virgule quand on écrit le résultat dans la variable entière `n3` et donc, en général, `n3*n3` n'est pas égal à `reste`. Ce n'est que lorsque `reste` est un carré parfait que le test est vrai et qu'on incrémente `compteur`. (Par exemple, si `reste=70`, `n3` vaudra 8 parce que $\sqrt{70} = 8,3666\dots$, et `n3*n3` ne vaut que 64. Si `reste=81`, alors `n3` vaut bien 9.)

Cette nouvelle version fait $(\sqrt{X})^2 = X$ opérations au lieu de $X^{3/2}$, ce qui est beaucoup mieux. On peut encore un tout petit peu améliorer les performances de cette fonction (exercice laissé au lecteur). Voici deux pistes : 1) si `reste` est négatif, ça ne sert à rien de tester des valeurs de `n2` encore plus grandes ! Il faut interrompre la boucle sur `n2` et passer à la valeur suivante de `n1`. (Voir l'instruction `break` dans le chapitre 5.) 2) On obtient plusieurs fois la même réponse, à une permutation près. On pourrait s'arranger pour ne tester que les nombres tels que $n_1 \leq n_2 \leq n_3$. Si on trouve le triplet (0,40,192), on sait qu'il y a au total 6 solutions avec ces trois nombres dans un ordre ou dans l'autre.

Chapitre 3

Types et variables

Toutes les variables du C doivent être déclarées avec leur type. Le type permet de savoir quelle est la place que la variable occupe en mémoire et quel genre de données la variable peut contenir. Le C connaît quelques types simples (comme les entiers ou les nombres à virgule), et peut également manipuler des types plus compliqués, comme les tableaux.

3.1 Types simples

Pour pouvoir représenter n'importe quel nombre entier ou n'importe quel nombre réel, il faudrait disposer d'une mémoire infinie. Tout ce que l'ordinateur peut représenter sont des approximations limitées des entiers ou des réels. Ainsi, les variables de type « entier » ne peuvent contenir que des nombres plus petits qu'une certaine valeur. Par exemple, le programme suivant essaye d'afficher les valeurs de $n!$ pour n variant de 0 à 19 :

```
#include <stdio.h>

int fact(int n) {                               // Calcule la factorielle de n
    int result = 1;
    while ( n > 1 ) {
        result = result * n;
        n = n - 1;
    }
    return result;
}

int main(void) {
    int i = 0;
    while (i < 20) {
        printf("La factorielle de %d est %d\n", i, fact(i));
        i = i + 1;
    }
    return 0;
}
```

compilé avec gcc sur un pentium sous linux, le résultat est

```
|| La factorielle de 0 est 1
|| La factorielle de 1 est 1
|| La factorielle de 2 est 2
|| La factorielle de 3 est 6
|| La factorielle de 4 est 24
|| La factorielle de 5 est 120
|| La factorielle de 6 est 720
|| La factorielle de 7 est 5040
|| La factorielle de 8 est 40320
|| La factorielle de 9 est 362880
|| La factorielle de 10 est 3628800
|| La factorielle de 11 est 39916800
|| La factorielle de 12 est 479001600
|| La factorielle de 13 est 1932053504
|| La factorielle de 14 est 1278945280
|| La factorielle de 15 est 2004310016
|| La factorielle de 16 est 2004189184
|| La factorielle de 17 est -288522240
|| La factorielle de 18 est -898433024
|| La factorielle de 19 est 109641728
```

Le résultat est mathématiquement correct jusqu'à $n = 12$ inclus. Au delà, toutes les valeurs sont fausses. On a même des nombres négatifs ! La raison est qu'une variable de type `int`, dans un programme compilé avec gcc sur un pentium sous linux, ne peut contenir que des valeurs entre -2 147 483 648 et 2 147 483 647 (bornes incluses). Lorsqu'une opération arithmétique dépasse une de ces bornes, le résultat aura une valeur différente de la valeur attendue.

Sur certains ordinateurs, il est même possible que le dépassement des bornes d'un type signé provoque un arrêt immédiat du programme. Une erreur de ce type est à l'origine de l'explosion de la fusée Ariane 5 en 1996. Voir <http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html> pour les détails.

Les variables de type réel ont les mêmes limitations que les variables de type entier, mais, en plus, elles ne peuvent contenir les nombres qu'avec une précision limitée. Considérons l'exemple

```
|| double x;
|| x = 1 + 1e-16 - 1;
```

Le type `double` est un type de nombre à virgule, et `1e-16` signifie 10^{-16} . Mathématiquement, `x` devrait valoir 10^{-16} mais, avec gcc sur un pentium sous linux, on obtient la valeur 0. La raison est que le type `double` peut représenter le nombre 1, peut représenter (à peu près) le nombre 10^{-16} , mais n'est pas assez précis pour distinguer $1 + 10^{-16}$ de 1. Le nombre $1 + 10^{-16}$ est donc représenté comme le nombre 1 et, fatalement, si on retranche 1, on obtient 0.

Pour ces raisons, on évite de dire que le type `double` permet de représenter un nombre réel. On préfère utiliser « nombre à virgule flottante », ou « nombre à virgule », ou « flottant ».

Au passage, « `x = 1 - 1 + 1e-16;` » donne bien 10^{-16} . Cela signifie donc que l'addition, en C, n'est pas associative !

Une grande difficulté du C est que les limites des différents types dépendent du processeur et du compilateur. Par exemple, la norme du C spécifie que le type `int` a « la taille naturelle que suggère l'architecture et l'environnement d'exécution ». Autrement dit, c'est en manipulant des `int` que le processeur sera le plus rapide, mais on ne sait pas *a priori* quelles sont les valeurs que l'on peut représenter dans un `int`.

3.1.1 Types entiers

Le type entier le plus utile est le type `int`. Sur la plupart des ordinateurs actuels, il permet de représenter des nombres entre moins deux milliards et plus deux milliards, mais ces limites ne sont pas garanties. Les compilateurs C pour MS-DOS avaient, typiquement, un type `int` allant de -32 768 à 32 767. Il est probable que les mêmes limites aient cours sur certains matériels embarqués, comme des cartes à puce. Sur certaines machines 64 bits, le type `int` peut avoir des limites beaucoup plus grandes, de l'ordre de $\pm 9 \cdot 10^{18}$.

La norme garantit qu'un `int` peut contenir un nombre entre -32 767 et 32 767.

Le type `long` ou `long int` est un type entier qui est au moins aussi long que `int`. En fait, sur un PC standard, ces deux types ont les mêmes limites.

La norme garantit qu'un `long` peut contenir un nombre entre -2 147 483 647 et 2 147 483 647.

Le type `long long` ou `long long int` est un type entier qui est au moins aussi long que `long`. Effectivement, sur un PC standard, ce type permet de stocker des nombres bien plus grand que `int`. Il faut néanmoins savoir que la manipulation de variables de type `long long` prend plus de temps qu'il n'en faut pour manipuler des `int`.

Le type `long long` a été introduit en C 99. La norme garantit qu'un `long long` peut contenir un nombre entre -9 223 372 036 854 775 807 et 9 223 372 036 854 775 807.

Le type `short` ou `short int` est plus petit que le type `int`, et ne permet donc pas de représenter des nombres aussi grand qu'un `int`. Il n'est utile que dans les applications où la place mémoire est limitée et où on a besoin de stocker *beaucoup* de *petits* nombres.

La norme garantit qu'un `short` peut contenir un nombre entre -32 767 et 32 767.

Enfin, le type `char` est le plus petit de tous les types entiers. Il est spécifiquement conçu pour représenter des *caractères*, c'est-à-dire les lettres, les ponctuations, les chiffres, etc. En effet, dans la mémoire de l'ordinateur, chaque caractère est représenté sous la forme d'un nombre entier. Un `char` est aussi la plus petite unité mémoire naturellement accessible par le processeur.

Pour tous ces types, il existe une version « `unsigned` », qui ne peut contenir que des entiers positifs ou nul. Par exemple, si l'on déclare les variables `i` et `j` comme

```
|| unsigned int i;  
|| int j;
```

alors, sur un PC standard, `i` peut contenir des nombres entre 0 et $4 \cdot 10^9$, alors que `j` peut contenir des nombres entre $-2 \cdot 10^9$ et $+2 \cdot 10^9$. Le nombre de valeurs différentes que `i` et `j` peuvent contenir est le même et `i` et `j` occupent donc la même place dans la mémoire de l'ordinateur.

Les types `unsigned` peuvent rendre de grand service, mais il faut se méfier ; le C a quelques comportements non intuitifs lorsque l'on compare des entiers signés et des entiers non signés. Par exemple :

```
|| unsigned int i = 100;  
|| int j = -200;  
|| if ( j < i ) {  
||     printf("oui\n");  
|| } else {
```

```
|| } printf("non\n");
```

va afficher « non » contrairement à ce qu'on pourrait s'attendre, alors qu'on aurait eu « oui » si `i` n'avait pas été `unsigned`.

La norme ne précise pas si le type `char` peut ou non accueillir des nombres négatifs. On peut employer les types `unsigned char` ou `signed char` si l'on veut être sûr dans un sens ou dans l'autre. Le type `char` est semblable soit au type `unsigned char`, soit au type `signed char`, mais on ne sait pas auquel. La norme garantit qu'un `signed char` peut contenir un nombre entre -127 et 127, et qu'un `unsigned char` peut contenir un nombre entre 0 et 255.

Dans l'entête `<stdint.h>` (introduite en C 99) peuvent être définis des types comme `int32_t` ou `uint64_t` qui représentent, respectivement un type entier signé de 32 bits et un type entier non signé de 64 bits.

Dans l'entête `<stddef.h>` sont définis des types comme `size_t` qui est le type entier le plus approprié pour représenter la place que des objets occupent dans la mémoire de l'ordinateur. Ce type est `unsigned` et est un bon type pour représenter des index de tableaux.

Dans ce cours, pour les types entiers, nous n'utiliserons que le type `int` (dans tous les cas courants) et le type `char` (pour représenter des caractères). Dans certains cas exceptionnels, on pourra utiliser le type `long long`.

3.1.2 Types de nombres à virgule flottante

Nous utiliserons dans ce cours le type `double`.

Il existe également le type `float`, moins précis que le type `double` mais prenant moins de place en mémoire, et le type `long double`, plus précis que le type `double` mais prenant plus de place.

Le type `long double` a été introduit en C 99.

3.1.3 Écritures de constantes

Une constante entière s'écrit tout naturellement en décimal. Par exemple

```
|| int i = -42;
```

On peut aussi l'écrire en octal ou en hexadécimal à condition de préfixer le nombre par un `0` ou par un `0x`. Ainsi, par exemple « `int i = 010;` » initialise `i` avec la valeur 8.

Il est également possible de spécifier que la constante est de type `unsigned` en ajoutant un `U` à la fin du nombre, et qu'elle est de type `long` ou `long long` en ajoutant `L` ou `LL` à la fin du nombre. Il est très rare que ces suffixes soient effectivement indispensables.

Une constante de type `char` est représenté par le caractère mis entre apostrophes. Par exemple :

```
|| char c = 'A';
```

Il y a quelques caractères spéciaux, comme par exemple, `'\n'` qui représente le caractère « retour à la ligne ».

Comme on l'a dit plus haut, le type `char` est en fait un type entier. Dans la mémoire de l'ordinateur, tous les caractères sont représentés par des nombres suivant un certain codage qui, en théorie, peut dépendre de l'ordinateur. En pratique, tous les ordinateurs sauf quelques pièces de musée utilisent le codage ASCII où, par exemple, la valeur de la lettre `A` est 65. Toutes les opérations (additions, comparaisons, etc.) que l'on peut faire sur des entiers peuvent donc être faites sur des `char`. Par exemple, après

```
|| char c = 'A';  
|| c = c + 1;
```

`c` vaudra la valeur 66, qui correspond à la lettre `B`.

Une constante de type réel s'écrit avec un point décimal ou en écriture « scientifique » avec la lettre `e` qui marque une puissance de 10.

```
|| double pi = 3.141593;  
|| double hbar = 1.0545716e-34;
```

On peut écrire une constante de type `float` ou de type `long double` en ajoutant un `F` ou un `L` à la fin du nombre. L'ordinateur sait manipuler des nombres de type `double` un peu spéciaux, comme $\pm\infty$, `NaN` (résultat indéterminé), ± 0 .

3.1.4 Conversions entre types simples

Un ordinateur sait additionner deux entiers ou deux nombres à virgule flottante, mais ne sait pas directement ajouter un entier à un nombre à virgule. Pour réaliser cette opération, il faut d'abord convertir l'entier en nombre à virgule. En C, cette opération de conversion est automatique :

```
|| double x = 3.7;  
|| int n = 6;  
||  
|| x = x + n;
```

Dans cet exemple, la variable `n` est convertie en `double`, puis l'addition est effectuée.

Parfois, l'opération s'accompagne d'une perte de précision :

```
|| double x = 3.7;  
|| int n;  
||  
|| n = x + 1;
```

Dans cet exemple, le 1 est d'abord converti en `double`, l'addition (entre deux `double`) est effectuée, et *le résultat est converti en `int`* avant d'être écrit dans la variable `n`. Comme un `int` ne peut pas contenir 4,7, la valeur finale de `n` est 4.

Convertir un nombre à virgule flottante en entier enlève donc la partie décimale du nombre.

Il existe un opérateur permettant de forcer la conversion : on écrit entre parenthèses le type que l'on désire obtenir avant l'expression que l'on désire transformer. Cet opérateur a une grande priorité. Ainsi, on peut écrire :

```
|| int n = (int)3.14;  
|| double x = (int)3.14;
```

Dans cet exemple, la première conversion forcée est inutile. Comme le nombre 3,14 est de toute façon écrite dans une variable entière, le compilateur aurait de toute façon fait tout seul la conversion même si on n'avait pas écrit le `(int)`. Après cette opération, `n` vaut donc 3. Dans la deuxième ligne de cet exemple, le 3,14 est d'abord converti en `int` à cause de la conversion forcée et se transforme donc en 3. Comme le résultat est mis dans une variable de type `double`, le 3 est retransformé en `double` par une conversion automatique. À la fin, `x` vaut 3,0.

Quand un flottant est converti en entier, l'entier est toujours compris entre le flottant et zéro. Ainsi, « `(int)-3.9` » vaut -3.

Le jeu des conversions automatiques donne parfois des résultats surprenants. Considérons par exemple

```
|| double x = 5 / 2;
```

Ici, 5 et 2 sont deux constantes de type `int`. La division est donc une division entière et le résultat est un entier valant 2. Cet entier est alors converti en `double` avant d'être écrit dans `x`, qui vaut donc finalement 2,0. Si on voulait que `x` ait la valeur 2,5, il aurait fallu écrire «`double x = 5.0 / 2;`» ou «`double x = (double)5 / 2;`». En effet, dans ce cas, le 5 est un `double`, donc le 2 est automatiquement converti en `double`. La division, entre deux `double` donne un `double` valant 2,5 et ce résultat est écrit dans `x`.

3.2 Tableaux et chaînes de caractères

3.2.1 Tableaux

Une variable de type *tableau* permet de contenir plusieurs (et, potentiellement, beaucoup) de valeurs ayant un type commun. Il y a donc des tableaux de type `int` (qui peuvent contenir plusieurs `int`), des tableaux de type `double`, etc. Ainsi, par exemple,

```
|| int table[ 5 ];
```

permet de déclarer que `table` est un tableau contenant *cinq* éléments de type `int`. Comme d'habitude, cette déclaration réserve dans la mémoire de l'ordinateur la place mémoire nécessaire pour stocker ces cinq `int`.

Dans un tableau à n éléments, chaque élément est indexé par un numéro qui va de 0 à $n - 1$. Ce point est très important, car il est une grande source d'erreur : *le C compte à partir de zéro. Le premier élément a le numéro 0 et le n -ième élément a , donc, le numéro $n - 1$.* Pour accéder à un élément particulier, on met le nom du tableau suivi de l'index de l'élément entre crochets. Par exemple :

```
|| int table[ 5 ];  
   table[ 0 ] = 0;      // table[ 0 ] est le premier élément  
   table[ 1 ] = 1;      // deuxième élément  
   table[ 2 ] = 4;      // troisième élément  
   table[ 3 ] = 9;      // quatrième élément  
   table[ 4 ] = 16;     // cinquième et dernier élément
```

Comme pour une variable normale, on peut initialiser le tableau au moment de la déclaration. Ainsi, l'exemple qui précède aurait pu s'écrire

```
|| int table[ 5 ] = { 0, 1, 4, 9, 16 };
```

Notez que dans ce cas, le nombre d'éléments (5) est redondant puisqu'il est facile de compter le nombre d'éléments entre accolades. On peut donc enlever le « 5 » de l'expression précédente et écrire

```
|| int table[ ] = { 0, 1, 4, 9, 16 };
```

On peut lire le contenu du tableau aussi facilement que l'on peut y écrire

```
|| x = table[ 1 ] + table[ 3 ];
```

Pour que cet exemple soit valide, il est essentiel que l'on ait, auparavant, initialisé le tableau ou écrit quelque chose dans «`table[ 1 ]`» et dans «`table[ 3 ]`». Sinon, le comportement est indéfini.

Bien entendu, l'index de l'élément que l'on veut accéder peut être une expression quelconque, pas seulement un nombre. Par exemple, si `i` est une variable entière, « `table[ i ]` » permet d'accéder à l'élément du tableau dont l'index est contenu dans la variable `i`.

Depuis C 99, il est possible de déclarer, dans des fonctions, des tableaux dont la taille n'est pas déterminée à la compilation. Par exemple

```
|| void f(int m) {  
||     int table[m];  
||     ...  
|| }
```

est maintenant valide. Le tableau `table`, local à la fonction `f()`, a une taille égale au paramètre `m` de la fonction. Cette taille est donc susceptible de varier entre deux appels à `f()`. Nous vous déconseillons d'utiliser cette construction tant que vous n'avez pas appris ce qu'est une allocation mémoire et un débordement de pile.

3.2.2 Chaînes de caractères

Une chaîne de caractères, c'est-à-dire une succession de caractères, est représentée en C par un tableau d'éléments de type `char`. La seule particularité des chaînes de caractères par rapport aux autres tableaux est qu'il existe une syntaxe spéciale permettant de représenter une chaîne constante : on l'écrit entre guillemets doubles. Par exemple, on peut écrire :

```
|| char greeting[ ] = "Bonjour !";
```

La convention du C est qu'une chaîne de caractères se termine toujours par un caractère nul (qui n'est pas la même chose que le caractère représentant le chiffre 0). L'exemple ci-dessus est donc complètement équivalent à

```
|| char greeting[ ] = { 'B', 'o', 'n', 'j', 'o', 'u', 'r', ' ', '!', 0 };
```

(Notez l'absence de guillemets simples autour du zéro final.)

Quand on manipule des chaînes de caractère, il ne faut pas oublier ce caractère nul :

```
|| char greeting[40] = "Bonjour !";
```

déclare un tableau de quarante caractères. Compte tenu du caractère nul, la variable `greeting` ne peut accueillir que trente-neuf caractères effectifs. Ici, on initialise `greeting` avec « `Bonjour !` », qui fait neuf caractères. On a donc réservé trente caractères de trop.

Une chaîne de caractères se manipule comme n'importe quel tableau. Par exemple, après l'instruction « `greeting[4] = 'e';` », la chaîne `greeting` vaut « `Bonjeur !` ».

S'il est facile de changer un caractère dans une chaîne, *on ne peut pas donner une nouvelle valeur à une chaîne de caractères avec un simple signe « = »*. Par exemple,

```
|| char greeting[40] = "Bonjour !";    // Déclaration et initialisation ; correct  
|| greeting = "Hello !";              // INCORRECT; l'opérateur = est incapable de  
||                                     // recopier une chaîne de caractères
```

Pour recopier une chaîne de caractères, il faut donc faire une boucle et copier les caractères un à un, ou utiliser les fonctions de la bibliothèque standard qui feront la boucle pour vous : `strncpy()` déclarée dans l'entête `<string.h>` (voir chapitre 7) ou, éventuellement, `snprintf()` dans `<stdio.h>` (voir chapitre 6).

3.2.3 Tableaux en deux dimensions

On peut déclarer en C des tableaux en deux dimensions. Ainsi

```
|| double matrice[ 2 ] [ 4 ] ;
```

déclare que `matrice` est un tableau de deux lignes et quatre colonnes. Il y a donc huit éléments (de type `double`) qui sont `matrice[0][0]`, `matrice[0][1]`, `matrice[0][2]`, `matrice[0][3]`, `matrice[1][0]`, `matrice[1][1]`, `matrice[1][2]` et `matrice[1][3]`. Autrement dit, dans cet exemple, le premier index varie entre 0 et 1 et le deuxième index entre 0 et 3.

On peut initialiser un tel tableau à la déclaration :

```
|| double matrice[ 2 ] [ 4 ] = { { 1.000, 1.001, 1.002, 1.003 },  
                                { 5.000, 5.001, 5.002, 5.003 } } ;
```

Après cette déclaration, par exemple, `matrice[0][2]` vaut 1,002 et `matrice[1][0]` vaut 5,000.

De la même manière, on peut manipuler des tableaux en trois dimensions ou plus.

3.2.4 Utilisations d'un tableau comme paramètre à une fonction

Si on essaye de déclarer des fonctions qui acceptent des tableaux comme arguments, il se trouve que ça va apparemment marcher comme on pourrait s'y attendre :

```
|| void f( int size, int t[ ] ) {  
    ....  
}  
  
int main(void) {  
    int tableau[ ] = { 15, 30, 45, 60 };  
    int taille = 4;  
  
    f( taille, tableau );  
    ...  
}
```

Dans cet exemple, on définit une fonction `f()` qui prend deux arguments, le premier est un entier et le deuxième est un tableau d'entiers. La fonction `main()` montre comment on peut appeler la fonction `f()`. Ce type de code est autorisé.

Cependant, certaines choses ne se passent pas comme d'habitude. D'habitude, lorsqu'on appelle une fonction avec des arguments, la fonction reçoit une copie de la valeur des arguments. Copier un entier, c'est facile, mais copier un tableau, ça peut prendre du temps si le tableau est long. Pour éviter de perdre son temps à faire cette copie, le C préfère passer à la fonction *l'adresse de l'emplacement mémoire où se trouve le tableau*. Avec cette adresse, la fonction peut sans difficulté lire les éléments du tableau. Mais elle peut, tout aussi facilement *écrire de nouvelles valeurs*. Ainsi, lorsque `main()` appelle la fonction `f()` avec « `f( taille, tableau );` », la fonction `f()` *ne peut pas* modifier la variable `taille` (parce qu'elle n'a reçu qu'une copie de la valeur contenue dans `taille` et ne sait pas où se trouve `taille` dans la mémoire de l'ordinateur), mais elle *peut* facilement changer le contenu du tableau `tableau`.

L'adresse d'un emplacement mémoire s'appelle, en C, un *pointeur*. Ici, le deuxième argument de la fonction `f()` est en fait un *pointeur vers un int*. Bien que la définition ci-dessus soit correcte, on préfère habituellement marquer explicitement le fait que cet argument est un pointeur et l'on emploie plutôt la syntaxe suivante :

```
|| void f( int size, int * t ) {  
    ....  
}
```

3.3 Autres types de variables

Il existe en C d'autres types de variables que nous n'utiliserons pas dans ce cours

- Les *struct* permettent de regrouper sous un même nom plusieurs variables de types différents. Par exemple

```

#include <stdio.h>

struct ville { // Définition du type struct ville
    double latitude;
    double longitude;
    int habitants;
    char nom[30];
}; // Jusque là, aucune variable n'a été déclarée

struct ville x; // On déclare x, de type struct ville, sans l'initialiser
struct ville y = { 44.83, -0.52, 220000, "Bordeaux" }; // On déclare et initialise y (les champs sont remplis dans l'ordre)

struct ville z = { // On déclare on initialise z
    .nom = "Paris",
    .latitude = 48.8,
    .longitude = 2.33,
    .habitants = 2200000
}; // Ce style d'initialisation est apparu en C99

int main(void) {
    if (z.habitants > y.habitants)
        printf("%s a plus d'habitants que %s\n", z.nom, y.nom);
    return 0;
}

```

- Les pointeurs représentent l'adresse de l'emplacement d'une variable dans la mémoire de l'ordinateur. (Ils *pointent* vers une variable.) On peut avoir des pointeurs vers des `int`, des `double`, des `struct`, etc. On peut même avoir des pointeurs vers des pointeurs, ou des pointeurs vers des fonctions. Par exemple,

```
|| int * x;
```

déclare que `x` est un pointeur vers un `int`. Il y a deux opérateurs pour manipuler les pointeurs; l'opérateur « `*` », dit de *déréférencement* qui permet de lire ou d'écrire dans la zone mémoire pointée par le pointeur, et l'opérateur « `&` », dit de *prise d'adresse*, qui permet de récupérer l'adresse mémoire d'une variable. Par exemple

```

|| int n = 3;
|| int * x;

x = &n; // x pointe vers l'adresse mémoire de n
printf("%d\n", *x) // affiche la valeur pointée par x, à savoir 3
*x = 5; // met 5 dans la zone mémoire pointée par x.
// Maintenant, n vaut 5

```

- Les `union`, qui permettent d'indiquer qu'une même zone mémoire est partagée par plusieurs valeurs. Par exemple

```

|| union number {
||     int n;
||     double x;
|| };
|| union number val;

```

Dans `val`, on peut soit stocker un entier, soit un flottant, mais pas les deux. Pour accéder à l'entier, on utilise « `val.n` », pour accéder au flottant, on utilise « `val.x` ». Cependant, puisque « `val.n` » et « `val.x` » occupent le même emplacement mémoire, on aura des résultats bizarres si on écrit dans « `val.n` » puis qu'on lit dans « `val.x` ». Les `union` s'utilisent surtout en combinaison avec les `struct`.

- Les `enum` permettent de définir des types finis. Par exemple

```

|| enum couleurs { rouge, vert, bleu }; // Déclaration du type
|| enum couleurs x; // La variable x est de type enum couleurs
x = rouge;
...
if (x == vert) {
    ...
}

```

La variable `x` peut donc prendre trois valeurs différentes nommées `rouge`, `vert` et `bleu`. En interne, en fait, tout se passe comme si `x` était de type `int`, que `rouge` était un synonyme pour 0, `vert` un synonyme pour 1 et `bleu` un synonyme pour 2.

Chapitre 4

Opérateurs

Dans l'instruction suivante,

```
|| printf("Le résultat est %d\n", 2*(1+x));
```

le deuxième argument « $2*(1+x)$ » de `printf()` est une *expression* qui permet de calculer une valeur. Cette expression est construite à l'aide des deux *opérateurs* « + » et « * » qui permettent, respectivement, de faire une addition et une multiplication. Un opérateur agit sur des *opérandes* ; ici, les opérandes de l'opérateur « + » sont 1 et x. L'opérateur calcule une valeur à partir de ses opérandes (ici, l'opérateur « + » fait, bien sûr, l'addition de 1 et de x) et cette valeur peut servir d'opérande à un autre opérateur. Ainsi, les opérandes de « * » sont 2 et le résultat de « $1+x$ ».

Certains opérateurs, comme l'opérateur « = », permettent de changer la valeur contenue dans une variable (ou dans un tableau ou dans un emplacement mémoire quelconque.)

Les opérateurs ont des priorités relatives. Par exemple, « * » est plus prioritaire que « + » et l'expression « $2 * 1 + x$ » est interprétée comme « $(2 * 1) + x$ » et non comme « $2 * (1 + x)$ ». Une paire de parenthèses permet toujours d'imposer l'ordre d'évaluation de son choix.

La plupart des opérateurs du C prennent deux opérandes, l'un placé avant et l'autre placé après l'opérateur. On dit que ces opérateurs sont *binaires*.

Il existe également une dizaine d'opérateurs qui ne prennent qu'un seul opérande. Ces opérateurs sont dits *unaires*. Par exemple, dans l'instruction « $x = -y$; », l'opérateur « - » est un opérateur unaire qui calcule l'opposé de son unique opérande, à savoir y. D'un point de vue logique, cet opérateur est différent de l'opérateur binaire « - » qui permet de calculer la différence entre deux nombres. Lorsqu'un opérateur unaire est placé avant son opérande (comme « - »), on dit que c'est un opérateur *préfixe*. Lorsqu'il est placé après, on dit que c'est un opérateur *postfixe*.

Enfin, le C dispose d'un opérateur *ternaire* ayant, donc, trois opérandes.

4.1 Ordre d'évaluation

Dans une expression composée de plusieurs opérateurs, les opérateurs les plus prioritaires sont évalués en premier et les moins prioritaires sont évalués en dernier. Cependant,

il n'y a en général *aucune garantie* sur l'ordre d'évaluation des opérandes. Par exemple, dans l'expression

```
|| z = f(3) + g(4) * h(5);
```

on sait que la multiplication entre « `g(4)` » et « `h(5)` » sera d'abord effectuée, puis que le résultat de la multiplication sera ajouté à « `f(3)` » et, qu'enfin, le résultat de l'addition sera affecté à `z`. Cependant, on ne sait pas dans quel ordre les fonctions `f()`, `g()` et `h()` seront appelées. S'il est impératif que ces fonctions soient appelées dans cet ordre, il faut écrire plusieurs instructions. Par exemple, en supposant que le type de retour des trois fonctions est `int`,

```
|| int val1 = f(3);
|| int val2 = g(4);
|| int val3 = h(5);
||
|| z = val1 + val2 * val3;
```

L'ordre de priorité des opérateurs du C est résumé dans le tableau suivant ; sur une même ligne, les opérateurs ont la même priorité. Les opérateurs écrits en petits correspondent à des notions avancées. Pour chaque opérateur, on suppose que `x`, `y` et `z` sont des expressions ayant un bon type pour être utilisées avec cet opérateur. « `(type)` » indique le nom d'un type écrit entre parenthèse.

Les	<code>x++ ; x-- ; x.membre ; x->membre</code>
plus	<code>-x ; !x ; (type)x ; ++x ; --x ; +x ; sizeof x ; sizeof(type) ; &x ; *x ; ~x</code>
prioritaires	<code>x*y ; x/y ; x%y</code>
	<code>x+y ; x-y</code>
	<code>x<<y ; x>>y</code>
	<code>x<y ; x>y ; x<=y ; x>=y</code>
	<code>x==y ; x!=y</code>
	<code>x&y</code>
	<code>x^y</code>
	<code>x y</code>
	<code>x&&y</code>
	<code>x y</code>
Les	<code>x?:y:z</code>
moins	<code>x=y ; x*=y ; x/=y ; x%=y ; x+=y ; x-=y ; x<<=y ; x>>=y ; x&=y ; x^=y ; x =y</code>
prioritaires	<code>x,y</code>

« `membre` » désigne un membre d'un `struct` ou d'un `union`.

On peut considérer qu'un appel de fonction (« `f(5)` ») ou la sélection d'un élément dans un tableau (« `table[8]` ») sont les emplois de deux autres opérateurs, « `( )` » et « `[ ]` ». Du point de vue de leur priorité, ces deux opérateurs seraient alors dans la première ligne du tableau. De même, depuis C99, on peut écrire une valeur constante de type tableau, `struct` ou `union` en donnant le type entre parenthèses et les valeurs des composantes entre accolades. Si on considère que cette construction est un opérateur, il serait également dans la première ligne du tableau.

4.2 Opérateurs arithmétiques

Le C dispose des quatre opérations courantes « `+` », « `-` », « `*` » et « `/` ».

Ces opérateurs sont binaires et fonctionnent sur les types entiers ou réels avec le sens habituel. Il faut néanmoins se méfier des débordements quand le résultat d'une opération ne peut pas être représenté (voir le chapitre 3 et l'exemple de la factorielle). Le type du résultat est le même que le type des opérandes : si on divise deux `double`, on obtient un `double` et si on divise deux `int`, on obtient un `int`. Ainsi, « 5 / 2 » vaut 2 (division entière), alors que « 5.0 / 2.0 » vaut 2,5 (division de flottants). Si les deux opérandes ne sont pas du même type, celui qui a le type le plus « petit » est automatiquement converti pour avoir le même type que l'autre opérande. Par exemple, si on multiplie un `double` par un `int`, l'`int` est d'abord converti en `double`.

Les règles précises de ces conversions automatiques sont horriblement compliquées et peuvent parfois donner des résultats inattendus, en particulier si on essaye de diviser un entier de type `unsigned` et un entier `signed`.

Les opérateurs « + » et « - » permettent aussi, sous certaines conditions, d'« additionner » un pointeur et un entier, et de « soustraire » deux pointeurs.

L'opération modulo « % » permet d'obtenir le reste de la division entre deux entiers. Par exemple, « 17 % 5 » vaut 2. On a toujours l'égalité « `a == (a / b) * b + (a % b)` ».

« `a % b` » est compris entre 0 et « `b-1` » seulement si « `a` » et « `b` » sont positifs. Quand on divise deux entiers, le résultat est l'entier que l'on obtient en enlevant la partie fractionnaire du rapport. Ainsi, « -20 / 11 », qui, algébriquement, vaut -1,90909..., donne -1 et non -2. Pour que l'égalité avec le modulo soit respectée, on doit donc avoir « `-20 % 11 == -9` ».

L'opérateur unaire « - » permet de changer le signe de son opérande. Par exemple, après « `x = -y` », la variable `x` contient l'opposé de `y`.

Par symétrie, il existe également l'opérateur unaire « + », qui ne fait rien. Par exemple : « `y = +x` ».

4.3 Opérateurs de comparaison

Les six opérateurs binaires de comparaison « == » (est égal à), « != » (est différent de), « < » (est strictement inférieur à), « > » (est strictement supérieur à), « <= » (est inférieur ou égal à) et « >= » (est supérieur ou égal à) permettent de comparer deux valeurs arithmétiques (entiers ou flottants). Le résultat de la comparaison est une valeur *booléenne* qui vaut « Vrai » ou « Faux » et qui permet, à l'aide d'instructions complexes comme « `if` » ou « `while` », d'altérer le déroulement du programme selon qu'une condition est ou non réalisée. Par exemple

```

| int answer = read_answer();
| while (answer != 42) {
|     printf("Perdu, essayez encore.\n");
|     answer = read_answer();
| }
| printf("Bravo, vous avez trouvé.\n");

```

Ces opérateurs de test ne permettent pas de comparer si deux chaînes de caractères ou deux tableaux sont identiques.

Par contre, ces opérateurs permettent, dans certains cas, de comparer des pointeurs.

Il faut faire attention à ne pas confondre l'opérateur de test « == » et l'opérateur d'affectation « = ».

Utiliser « == » et « != » peut s'avérer délicat avec les nombres à virgule. Par exemple

```

|| double x = sqrt(3);
|| if ( x*x == 3 ) {
||     /* ... */
|| }

```

`sqrt()` est une fonction définie dans l'entête `<math.h>` qui calcule la racine carrée (*SQuare RooT*) de son argument. On s'attend donc à ce que `x*x` vaille 3, mais il s'avère que ce n'est pas forcément le cas, à cause de la précision finie des `double`. Avec gcc sur un PC linux, `x` vaut 1,732 050 807 568 877 193... alors que $\sqrt{3}$ vaut 1,732 050 807 568 877 294... , et `x*x` vaut 2,999 999 999 999 999 556... qui est *différent* de 3. Pour comparer deux nombres flottants, il vaut mieux se donner une marge d'erreur. Par exemple :

```

|| double epsilon = 1e-10;
||
|| double x = sqrt(3);
|| if ( fabs(x*x - 3) < epsilon ) {
||     /* ... */
|| }

```

La fonction `fabs()`, définie dans `<math.h>`, calcule la valeur absolue d'un nombre flottant. Au lieu de regarder si `x*x` et 3 sont égaux, l'exemple qui précède regarde si la différence entre `x*x` et 3 est, en valeur absolue, inférieure à 10^{-10} .

Comme pour les opérateurs arithmétiques, il y a une conversion automatique si on essaye de comparer deux expressions ayant des types différents, et cette conversion peut être source de problèmes si on essaye de comparer un entier signé avec un entier non signé.

En interne, une valeur booléenne est tout simplement un entier avec la convention que « 0 » signifie « Faux » et que n'importe quoi d'autre signifie « Vrai ». Ainsi,

```

|| if (x) {
||     /* ... */
|| }

```

se comporte exactement comme

```

|| if (x != 0) {
||     /* ... */
|| }

```

Les opérateurs de comparaison renvoient « 1 » pour « Vrai » et, bien sûr, « 0 » pour « Faux ».

4.4 Opérateurs booléens

Les *opérateurs booléens* permettent de faire des opérations sur les deux valeurs booléennes « Vrai » et « Faux ». Il y a deux opérateurs binaires, « && » et « || » qui, respectivement, signifient « ET » et « OU », et un opérateur unaire, « ! » qui exprime la négation. Ainsi, par exemple,

```

|| if ( x == 3 && y < 5 ) {
||     ...
|| }

```

se lit « SI (x est égal à 3 ET y est inférieur à 5), ALORS (exécuter les lignes entre accolades). »

L'opérateur « && » (ET) donne une valeur « Vrai » si et seulement si *ses deux opérandes* sont eux-mêmes des « Vrai ». L'opérateur « || » (OU) donne une valeur « Vrai » si et seulement si *au moins l'un de ses opérandes* est « Vrai ». Autrement dit,

« Vrai » && « Vrai »	vaut	« Vrai »		« Vrai » « Vrai »	vaut	« Vrai »
« Vrai » && « Faux »	vaut	« Faux »		« Vrai » « Faux »	vaut	« Vrai »
« Faux » && « Vrai »	vaut	« Faux »		« Faux » « Vrai »	vaut	« Vrai »
« Faux » && « Faux »	vaut	« Faux »		« Faux » « Faux »	vaut	« Faux »

L'opérateur « ! » de négation transforme un « Vrai » en « Faux » et réciproquement.

! « Vrai »	vaut	« Faux »
! « Faux »	vaut	« Vrai »

Par exemple,

```

|| if ( !(x>5) ) {
||     ...
|| }

```

se lit « SI (IL EST FAUX QUE x est supérieur à 5) ALORS (exécuter les lignes entre accolades) ». C'est donc complètement équivalent à

```

|| if ( x<=5 ) {
||     ...
|| }

```

Les opérateurs booléens jouent un rôle important en mathématique et en programmation et il est utile de se familiariser avec leurs propriétés.

Les deux opérateurs « && » et « || » ont, en C, une propriété unique : il est *garanti* que l'opérande de gauche est évalué *en premier* et que l'opérande de droite n'est évalué *que si c'est nécessaire* : avec l'opérateur « && », si le premier opérande est « Faux », on n'a pas besoin d'évaluer le second opérande pour savoir que le résultat est « Faux ». Avec l'opérateur « || », si le premier opérande est « Vrai », on n'a pas besoin d'évaluer le second opérande pour savoir que le résultat est « Vrai ». Cette propriété est très importante et très utile dans des expressions comme

```

|| if ( x==0 || 100/x > y ) {
||     ...
|| }

```

Dans ce cas, si « x==0 », le test est « Vrai » (les instructions entre accolades sont exécutées) et l'ordinateur ne va pas regarder si « 100/x » est ou non supérieur à « y ». C'est une bonne chose puisqu'essayer de diviser cent par zéro pourrait poser des problèmes allant jusqu'à l'arrêt du programme. De même, dans une expression comme

```

|| if ( f()==0 && g()==0 ) {
||     ...
|| }

```

l'ordinateur va d'abord appeler la fonction f() puis, *uniquement si sa valeur de retour est zéro*, appeler la fonction g().

4.5 Opérateurs d'affectation

L'opérateur binaire « = » permet d'écrire une valeur dans une zone mémoire, c'est-à-dire, en général, dans une variable ou dans un élément de tableau. Par exemple,

```
|| x = 1+2;
```

écrit « 3 » dans la variable « x ».

Une zone mémoire peut être décrite par un nom de variable, un élément de tableau, un élément d'un **struct** ou d'un **union**, ou un pointeur déréférencé par l'opérateur unaire « * ». Cette zone mémoire peut être décrite par une expression compliquée, mais toutes les expressions ne décrivent pas une zone mémoire : par exemple, une expression comme « f(3)=x; » n'a aucun sens en C.

Quand c'est nécessaire, une conversion automatique est réalisée : si on essaye d'écrire un entier dans une variable de type **double**, l'entier est d'abord converti en flottant.

Une expression comme « x = 1+2 » peut être évaluée et faire partie d'une expression plus complexe. Sa valeur est la valeur qui vient d'être écrite dans la zone mémoire, ici : « 3 ». Cette propriété permet d'écrire des choses comme « y = x = 1+2; »

Un opérateur d'affectation peut changer la valeur d'un élément d'un tableau, mais ne permet pas de changer d'un coup un tableau (ou une chaîne de caractères) tout entier.

Les cinq opérateurs binaires « *= », « += », « -= », « /= » et « %= » permettent de combiner une opération arithmétique et une affectation. Par exemple,

```
|| x *= 5;
```

multiplie la variable « x » par 5 et écrit le résultat dans « x ». Dans ce cas simple, on aurait pu écrire, de manière équivalente, « x = x * 5; ». Les opérateurs « += », « -= », « /= » et « %= » fonctionnent sur le même modèle.

Il y a une différence entre « x *= 5; » et « x = x*5; » si « x » est une expression plus complexe qu'une simple variable. Par exemple

```
|| table[ f(3) ] *= 5;
```

pour savoir quelle zone mémoire doit être multipliée par 5, il faut appeler la fonction f() pour évaluer f(3). Dans l'expression qui précède, la fonction f() est appelée *une fois*. Si on avait écrit

```
|| table[ f(3) ] = table[ f(3) ] * 5;
```

la fonction f() aurait été appelée *deux fois*. Si cette fonction a un effet de bord (par exemple, si elle affiche quelque chose sur l'écran), le comportement du programme n'est pas le même dans les deux cas.

Toujours sur le même modèle, il existe également les opérateurs « <<= », « >>= », « &= », « ^= » et « |= » qui font des opérations bit à bit dans une zone mémoire (voir la section 4.6.7).

Les opérateurs unaires « ++ » (opérateur d'incréméntation) et « -- » (opérateur de décréméntation) permettent, respectivement, d'ajouter ou de retrancher 1 à une zone mémoire. Ces deux opérateurs peuvent s'employer en notation préfixe ou postfixe. Ainsi, les expressions « x ++ » ou « ++ x » permettent de rajouter 1 à la variable « x ».

La différence entre la forme préfixe et la forme postfixe est que la valeur de « x ++ » est la valeur de « x » *avant* l'incréméntation alors que la valeur de « ++ x » est la valeur de « x » *après* l'incréméntation. Par exemple,

```
|| x = 3;
|| table[ x ++ ] = 0;
```

écrit la valeur 0 dans « table[3] » alors que

```
|| x = 3;
|| table[ ++ x ] = 0;
```

écrit la valeur 0 dans « table[4] ». Dans les deux cas, « x » vaut 4 à la fin du fragment.

En utilisant, dans une unique expression, plusieurs opérateurs d'affectation, on peut concevoir des expressions dont le résultat dépendrait de l'ordre d'évaluation. Toutes ces expressions sont illégales. Par exemple

```

|| int i = 5;
|| table [ i ] = i++; // Expression ILLÉGALE

```

L'opérande de droite vaut 5, mais l'opérande de gauche peut représenter la zone mémoire « `table[ 5 ]` » ou « `table[ 6 ]` » selon qu'il a été évalué avant ou après l'opérande de droite.

4.6 Autres opérateurs

4.6.1 Opérateurs de changement de type

Un type écrit entre parenthèses est un opérateur unaire préfixe qui permet de convertir une valeur pour qu'elle ait le type indiqué. Par exemple,

```
(int)2.8
```

est une valeur entière (plus précisément, de type `int`) valant 2.

4.6.2 Opérateur de mesure de la place mémoire

L'opérateur unaire préfixe `sizeof` permet de connaître la place occupée en mémoire par une variable d'un type donné. L'opérande de `sizeof` peut être soit le nom d'un type écrit entre parenthèses, soit une expression quelconque ayant le type dont on veut déterminer la taille. Par exemple, si `x` est une variable de type `double`, « `sizeof x` » et « `sizeof (double)` » sont équivalents et donnent la place occupée en mémoire par un `double`. Cette place mémoire est toujours un nombre entier, et l'unité est la place mémoire occupée par un `char`. Autrement dit, on a toujours « `sizeof (char) == 1` ».

4.6.3 Opérateurs d'adresse et de déréférencement

Les opérateurs unaires préfixes « `&` » (prise d'adresse) et « `*` » (déréférencement) servent à manipuler des pointeurs.

L'opérateur « `&` » permet de déterminer l'adresse d'une zone mémoire. Le résultat est alors un pointeur vers le type contenu dans cette zone mémoire. Par exemple, si « `x` » est une variable de type `double`, alors « `&x` » est une valeur de type « `double *` » (c'est-à-dire pointeur vers un `double`) qui contient l'adresse de l'endroit où, dans la mémoire de l'ordinateur, est stocké la variable « `x` ». Comme pour les opérateurs d'assignement, on peut appliquer l'opérateur « `&` » sur des expressions arbitrairement compliquées à condition que ces expressions représentent une zone mémoire.

L'opérateur « `*` » doit avoir un opérande de type pointeur et permet d'accéder à la zone mémoire contenue dans l'opérande, pour lire ou modifier la valeur qui y est inscrite. Si la variable « `p` » est de type « `double *` », alors « `*p` » est de type `double`.

Exemple :

```

double * p;
double * q; // p et q sont des pointeurs vers un double

double x, y;

p = &x; // p pointe sur x
*p = 3.14; // on vient de changer, via p, la valeur de x.

q = &y; // q pointe sur y
*q = *p; // on vient de changer, via p, la valeur de y.
// y contient désormais la valeur pointée par p, à savoir 3.14

q = p; // on vient de changer la valeur de q. Maintenant, q
// pointe sur la même zone mémoire que p, à savoir sur x.
// Ni x ni y n'ont été changés par cette dernière affectation.

```

4.6.4 Membres d'un struct ou d'un union

L'opérateur « . » permet de désigner un membre donné d'un **struct** ou d'un **union** afin de pouvoir lire ou modifier la valeur de ce membre. Par exemple

```
|| struct couleur {
||     double rouge;
||     double vert;
||     double bleu;
|| };
||
|| struct couleur x;      // x est de type struct couleur
||
|| x.rouge = 0.87;
|| x.vert  = 0.12;
|| x.bleu  = 0;
||
|| double luminosite = (x.rouge + x.vert + x.bleu) / 3;
```

L'opérateur « -> » sert à désigner un membre donné d'un **struct** ou d'un **union** contenu dans une zone mémoire désignée par un pointeur de type approprié. Pour continuer l'exemple précédent,

```
|| struct couleur * p;    // p est un pointeur sur un struct couleur
|| ...
|| p->vert = 0.25;
```

Si « p » est un pointeur vers un **struct couleur**, alors « p->vert » est équivalent à « (*p).vert ».

4.6.5 Opérateur conditionnel

L'opérateur conditionnel « ? : » est l'opérateur ternaire du C. Il s'emploie sous la forme « **exp1** ? **exp2** : **exp3** » où **exp1** est une expression ayant une valeur booléenne. Si **exp1** vaut « Vrai », alors **exp2** est évalué et le type et la valeur de l'expression sont ceux de **exp2** et, sinon, **exp3** est évalué et le type et la valeur de l'expression sont ceux de **exp3**. Par exemple

```
|| z = 8 + (op == 1 ? x + y : x*y);
```

Si « op » vaut 1, alors z prend la valeur « 8 + (x + y) » et, sinon, z prend la valeur « 8 + x*y ».

4.6.6 Opérateur virgule

L'opérateur binaire « , » permet d'effectuer deux actions dans une unique expression. L'opérande de gauche est *d'abord* évalué, puis l'opérande de droite est évalué. Le type et la valeur de l'expression sont ceux de l'opérande de droite.

En pratique, cet opérateur n'est utile que dans les boucles **for**. Par exemple :

```
|| for (i=0, j=0; i < 5; i++, j+=i) { ... }
```

4.6.7 Opérateurs de manipulation de bits d'un entier

Les types entiers sont représentés en binaire, c'est-à-dire qu'un nombre entier est codé comme une suite de 0 et de 1. Ces chiffres binaires s'appellent des bits. Les opérateurs binaires « & », « | » et « ^ » permettent de combiner deux à deux les bits de deux entiers selon les opérations logiques « ET », « OU » et « OU EXCLUSIF ». Par exemple, si on considère les deux nombres binaires 1100 et 1010, on a

<u>1100</u> & <u>1010</u>	vaut	<u>1000</u>
<u>1100</u> <u>1010</u>	vaut	<u>1110</u>
<u>1100</u> ^ <u>1010</u>	vaut	<u>0110</u>

(On a considéré ici des nombres de quatre bits. En pratique, les entiers des ordinateurs ont plus que quatre bits. On peut, si l'on veut, supposer, que l'on a juste écrit les quatre premiers bits et que tous les autres valent 0.)

$\boxed{1100}$ n'est pas le nombre mille cent, mais l'entier dont les bits successifs sont 1, 1, 0 et 0. En l'occurrence, cet entier est le nombre douze.

Avec l'opération « & » (ET), un bit du résultat est mis à 1 si et seulement si *les deux bits* correspondants des opérandes valent 1. Avec l'opération « | » (OU), un bit du résultat est mis à 1 si et seulement si *au moins l'un* des deux bits correspondants des opérandes vaut 1. Avec l'opération « ^ » (OU EXCLUSIF), un bit du résultat est mis à 1 si et seulement si *exactement un* des deux bits correspondants des opérandes vaut 1.

L'opérateur unaire préfixe « ~ » permet d'inverser tous les bits d'un entier : les 0 sont changés en 1 et les 1 en 0. Par exemple, si on considère un entier de huit bits,

$$\sim \boxed{01011100} \quad \text{vaut} \quad \boxed{10100011}$$

Les opérateurs binaires « >> » (décalage à droite) et « << » (décalage à gauche) permettent de décaler les bits d'un entier. L'opérande de gauche donne l'entier qui doit être décalé et l'opérande de droite indique de combien de bits l'entier doit être décalé. Par exemple,

$$\begin{array}{l} \boxed{01011100} \gg 3 \quad \text{vaut} \quad \boxed{00001011} \\ \boxed{01011100} \ll 1 \quad \text{vaut} \quad \boxed{10111000} \end{array}$$

Il vaut mieux utiliser ces opérateurs sur des entiers non signés ou, éventuellement, sur des entiers signés que l'on sait être positifs.

Chapitre 5

Opérations de contrôle : tests, boucles et sauts

Normalement, les instructions qui composent une fonction en C sont *toutes* exécutées *dans l'ordre* où elles sont écrites. Les opérations de contrôle permettent de modifier ce flux des instructions : un test (« `if` ») permet de n'exécuter une instruction ou un bloc d'instructions que si une certaine *condition* est remplie, une boucle (« `while` », « `for` » ou « `do` ») permet d'exécuter la même instruction ou le même bloc d'instructions plusieurs fois, tant qu'une certaine *condition* est remplie, et un saut (« `goto` », « `break` » ou « `continue` ») permet de « sauter » de manière inconditionnelle à une instruction donnée et de reprendre le flux normal à partir de cette instruction, sans prendre en compte les instructions présentes entre le point de départ et le point d'arrivée du saut. Enfin, l'instruction « `switch` », à mi-chemin entre le test et le saut, permet de sélectionner une portion de code à exécuter selon la valeur d'une variable.

5.1 Tests

La syntaxe est

```
|| if (condition) {  
||     série d'instructions  
|| } // La série d'instructions sera peut-être exécutée.  
|| suite du programme
```

ou

```
|| if (condition) {  
||     série d'instructions  
|| } else {  
||     autre série d'instructions  
|| } // Soit l'une, soit l'autre des deux séries sera exécutée.  
|| suite du programme
```

Dans les deux cas, la *condition* est évaluée. Si le résultat est « Vrai », la *série d'instructions* est exécutée, puis on passe à la *suite du programme*. Si le résultat est « Faux », l'*autre série d'instructions* est exécutée (s'il y a un bloc « `else` »), puis on passe à la

suite du programme. En aucun cas, la *série d'instructions* et l'*autre série d'instructions* peuvent être toutes les deux exécutées.

La *condition* qui contrôle le test est normalement une expression booléenne construite à l'aide des opérateurs de comparaison (« == », « != », « < », « > », « <= » et « >= ») et des opérateurs booléens (« && », « || » et « ! »). Cependant, en C, on peut également utiliser n'importe quelle expression arithmétique (nombre à virgule ou nombre entier) avec la convention que 0 signifie « Faux » et n'importe quoi d'autre signifie « Vrai ». Par exemple

<pre> if (x) { // identique à série d'instructions } </pre>	<pre> <i>if</i> (x != 0) </pre>
<pre> if (!x) { // identique à // ce qui est la même chose que série d'instructions } </pre>	<pre> <i>if</i> (!(x != 0)) <i>if</i> (x == 0) </pre>
<pre> if (!x && y>0) { // identique à série d'instructions } </pre>	<pre> <i>if</i> (x == 0 && y>0) </pre>

Un pointeur peut également être utilisé à la place d'une valeur booléenne. Tous les pointeurs sont interprétés comme « Vrai », sauf le pointeur « NULL » qui est interprété comme « Faux ». Un pointeur « NULL » ne peut jamais désigner une zone mémoire et ne doit donc jamais être déréférencé. Le symbole « NULL » est défini dans les entêtes `<stdlib.h>` et `<stddef.h>`.

Lorsqu'une *série d'instructions* est réduite à une unique instruction, on peut omettre les accolades qui l'entourent. Par exemple

<pre> if (x) printf("x n'est pas nul et vaut %d.\n",x); else printf("x vaut zéro.\n"); <i>suite du programme</i> </pre>

(On peut enlever les accolades, mais il faut laisser le point-virgule après chaque instruction.)

Une instruction complexe, comme un test (« `if (...) {...} else {...}` »), ou une boucle (« `while (...) {...}` »), compte pour une seule instruction. Cela permet d'écrire des choses comme

<pre> if (x==1) { printf("As"); } else if (x==11) { printf("Valet"); } else if (x==12) { printf("Dame"); } else if (x==13) { printf("Roi"); } else { printf("%d", x); } </pre>

Dans cet exemple, toute la partie qui va de « `if (x==11)` » jusqu'à la fin constitue le bloc (sans ses accolades) du premier `else`. Tout ce qui va du « `if (x==12)` » jusqu'à la fin constitue le bloc du deuxième `else`, etc. Au final, si on remet toutes les accolades, ce fragment de code est équivalent à

```

|
|  if (x==1) {
|      printf("As");
|  } else {
|      if (x==11) {
|          printf("Valet");
|      } else {
|          if (x==12) {
|              printf("Dame");
|          } else {
|              if (x==13) {
|                  printf("Roi");
|              } else {
|                  printf("%d", x);
|              }
|          }
|      }
|  }
|
|  }
|
|  }

```

La première version est quand même bien plus lisible, même si, dans ce cas particulier on aurait peut-être dû utiliser un `switch` (voir plus bas).

5.2 Boucles

Les boucles permettent d'exécuter une *série d'instructions* entre accolades tant qu'une certaine *condition* vaut « Vrai ». Comme dans le cas des tests, la *condition* est une expression booléenne ou une valeur arithmétique (interprétée comme « Vrai » si et seulement si la valeur est non nulle). Comme dans le cas des tests, les accolades peuvent être omises si la *série d'instructions* se réduit à une unique instruction.

5.2.1 La boucle « while »

La syntaxe est

```

|
|  while (condition) {
|      série d'instructions
|  }
|
|  suite du programme
|

```

Tant que la *condition* vaut « Vrai », la *série d'instructions* est exécutée. La *condition* est évaluée *avant* chaque exécution de la *série d'instructions*. Dès que la *condition* vaut « Faux », la boucle s'arrête et l'on passe à la *suite du programme*. En particulier, si la *condition* vaut « Faux » dès le départ, la *série d'instructions* n'est jamais exécutée.

5.2.2 La boucle « do »

La syntaxe est

```
|| do {  
||     série d'instructions  
|| } while (condition);           // ne pas oublier le point-virgule  
|| suite du programme
```

La *série d'instructions* est exécutée tant que la *condition* vaut « Vrai ». La *condition* est évaluée *après* chaque exécution de la *série d'instructions*. La *série d'instructions* est donc exécutée *au moins une fois*. Dès que la *condition* vaut « Faux », la boucle s'arrête et l'on passe à la *suite du programme*.

5.2.3 La boucle « for »

La syntaxe est

```
|| for (initialisation ; condition ; expression de mise à jour) {  
||     série d'instructions  
|| }  
|| suite du programme
```

L'*initialisation* est d'abord exécutée puis, tant que la *condition* vaut « Vrai », la *série d'instructions* puis l'*expression de mise à jour* sont exécutées. La *condition* est évaluée *avant* chaque exécution de la *série d'instructions* et l'*expression de mise à jour* est évaluée *après* chaque exécution de la *série d'instructions*. L'*initialisation* n'est exécutée qu'une seule fois, au tout début. Dès que la *condition* vaut « Faux », la boucle s'arrête et l'on passe à la *suite du programme*. En particulier, si la *condition* vaut « Faux » dès le départ, la *série d'instructions* et l'*expression de mise à jour* ne sont jamais exécutées.

Si l'instruction de saut « **continue** » n'est pas utilisée dans la boucle (voir plus loin), et hormis quelques cas pathologiques apparus en C99, la boucle « **for** » peut toujours se réécrire à l'aide d'une boucle « **while** » de la manière suivante :

```
|| initialisation ;  
|| while (condition) {  
||     série d'instructions  
||     expression de mise à jour ;  
|| }  
|| suite du programme
```

Habituellement, on emploie la boucle « **for** » pour exécuter une *série d'instructions* un nombre déterminé de fois, lorsqu'une certaine variable passe par toutes les valeurs entre un point de départ et un point d'arrivée. Cette variable s'appelle alors un compteur : l'*initialisation* sert à donner la valeur de départ du compteur, la *condition* sert à vérifier que le compteur n'a pas atteint sa valeur finale et l'*expression de mise à jour* permet de passer à la valeur suivante du compteur. Ainsi, l'exemple suivant met à zéro tous les éléments d'un tableau :

```
|| int table [50];  
|| int i;
```

```

|| for ( i = 0 ; i < 50 ; i++ )           // au lieu de i++, on aurait pu écrire i = i + 1
||     table[i] = 0;

```

(Comme il n'y avait qu'une seule instruction dans la boucle, on a enlevé les accolades.) L'instruction « `table[i] = 0` » est exécutée 50 fois au total pour `i` valant 0, puis 1, puis 2, etc. jusqu'à 49 inclus. L'*expression de mise à jour* (« `i++` ») est alors exécutée une dernière fois, `i` devient égal à 50, la *condition* (« `i < 50` ») n'est plus réalisée et la boucle s'arrête. Après l'instruction « `for` » de cet exemple, la variable `i` vaut donc 50.

Depuis C99, il est possible de déclarer le compteur dans la partie *initialisation* de la boucle. Ainsi, l'exemple précédent aurait pu être écrit

```

|| int table [50];
|| for ( int i = 0 ; i < 50 ; i++ )
||     table[i] = 0;

```

L'avantage est qu'ainsi, la variable « `i` » n'est plus définie après l'exécution de la boucle, et on ne risque pas de réutiliser cette variable par erreur. Malheureusement, au moment où ce document est écrit, gcc (version 3.4.2) n'accepte pas encore par défaut cette construction : il faut mettre explicitement l'option « `-std=c99` » sur la ligne de commande si on veut l'utiliser.

L'*initialisation*, la *condition* et l'*expression de mise à jour* sont optionnelles. Si la *condition* est omise, tout se passe comme si elle valait « Vrai ». Par exemple

```

|| for ( ; ; )
||     printf("infini\n");

```

affiche en boucle et indéfiniment le mot « infini ». Le même fragment aurait pu être écrit

```

|| while (1)
||     printf("infini\n");

```

Si on a absolument besoin de faire plusieurs choses dans l'*initialisation* ou l'*expression de mise à jour*, et qu'on est convaincu que le programme ne serait pas plus clair écrit autrement, on peut utiliser l'opérateur virgule (voir paragraphe 4.6.6). Par exemple

```

|| int i, total;
|| for ( i=0 , total=0 ; i < 10 && total < 200 ; i++ , total += taille(i) ) {
||     ...
|| }

```

5.3 Sauts

5.3.1 Le saut « goto »

Un « `goto` » permet de sauter des instructions sans les exécuter et de reprendre l'exécution à partir d'un point donné. Par exemple

```

||     goto labas;
||     printf("Je suis ici\n"); // cette instruction n'est pas exécutée
|| labas: // « labas » est un label
||     printf("Je suis là-bas\n"); // après le goto, l'exécution reprend à cet endroit

```

On indique l'endroit où l'exécution doit reprendre à l'aide d'un *label*, qui est un identificateur (« `labas` », dans l'exemple) suivi d'un deux-points placé juste avant une instruction.

Il doit y avoir une instruction après un label. En particulier, on ne peut pas mettre de label juste avant une accolade fermante. Cependant, l'instruction qui suit le label peut être l'instruction vide, c'est-à-dire un point-virgule mis tout seul.

L'usage du « `goto` » doit être parcimonieux et murement réfléchi. Les débutants en programmation ont parfois tendance à user et abuser du « `goto` », ce qui a tendance à rendre les programmes illisibles, incompréhensibles et peu efficaces. Dans la très grande majorité des cas, là où l'on veut utiliser un « `goto` » on peut et l'on *doit* utiliser un test (« `if` ») ou une boucle (« `for` », « `do` » ou « `while` »). Pour ces raisons, beaucoup de personnes, dont des programmeurs expérimentés, affirment qu'il ne faut *jamaïs* utiliser les « `goto` » en C. Même si c'est un peu dogmatique, ce n'est pas un mauvais conseil.

Bien qu'il soit toujours possible de s'en passer, il y a certains cas où un « `goto` » bien pensé peut permettre d'améliorer la lisibilité d'un programme. Voici un exemple :

```

| #define MAXESSAIS 5
|     int res;
|     int nombre;
|
|     int essais;
|     for(essais=0; essais < MAXESSAIS; essais++) {
|         printf("Tapez un nombre suivi de la touche « Entrée » : ");
|         res = scanf("%d", &nombre);
|         if (res == EOF) exit(1);
|         if (res == 1) goto ok;
|         printf("Désolé, votre entrée est incorrecte. Essayez encore\n");
|         scanf("%*[^\\n]"); // on jette la ligne invalide
|     }
|     printf("Vous êtes incapable de rentrer un nombre en %d essais.\n", essais);
|     exit(1);
| ok:
|     // nombre contient maintenant la valeur tapée au clavier

```

Pour des détails sur le fonctionnement de `scanf()` ou sur la signification de EOF, voir le chapitre 6.

5.3.2 Le saut « continue »

L'instruction « `continue` » permet de passer immédiatement à l'itération suivante dans une boucle « `while` », « `do` » ou « `for` ». Un « `continue` » est tout simplement équivalent à un « `goto` » qui irait *juste avant* l'accolade fermante de la boucle en question. Lorsque plusieurs boucles sont imbriquées les unes dans les autres, un « `continue` » ne permet d'agir que sur la boucle la plus *interne*.

L'exemple suivant affiche 50 nombres entre 0 et 99 qui ne sont pas des multiples de 3 :

```

| int n = 0;
| while ( n < 50 ) {
|     int x = rand() % 100;
|     if (x % 3 == 0) // On ne veut pas de multiples de 3.
|         continue; // Si on en a un, on retire .
|
|     printf("Le nombre %d n'est pas divisible par 3.\n", x);
|
|     n++;
|
|     // Le continue est équivalent à un saut vers ce point
| }

```

On peut remarquer qu'on *ne peut pas* reformuler directement l'exemple précédent avec une boucle « `for` » : si on écrit

```

| int n;

```

```

for ( n = 0 ; n < 50 ; n++ ) {
    int x = rand() % 100;
    if (x % 3 == 0)           // On ne veut pas de multiples de 3.
        continue;          // Si on en a un, on retire.

    printf("Le nombre %d n'est pas divisible par 3.\n", x);
                                // Le continue est équivalent à un saut vers ce point
}

```

on obtient un fragment de programme qui affiche *au plus* 50 nombres. En effet, dans les deux cas, le « `continue` » équivaut à un saut juste avant l'accolade fermante de la boucle. Dans le cas du « `while` », cela signifie qu'en particulier l'instruction « `n++`; » n'est pas exécutée : la valeur de `n` n'évolue pas tant qu'on n'a pas tiré de `x` satisfaisant. Dans le cas du « `for` », l'instruction « `n++`; » est de toute façon exécutée à *chaque* itération, même si `x` était divisible par 3 et que le « `continue` » a été exécuté. En résumé, la version avec la boucle « `while` » fait autant de tirages qu'il faut pour pouvoir afficher 50 nombres qui ne sont pas divisibles par 3, alors que la version avec la boucle « `for` » fait exactement 50 tirages et affiche les nombres qui, parmi les 50 tirés, ne sont pas divisibles par 3.

5.3.3 Le saut « `break` »

L'instruction « `break` » permet de sortir immédiatement d'une boucle « `while` », « `do` » ou « `for` », ou d'une instruction « `switch` » (voir plus bas). Un « `break` » est tout simplement équivalent à un « `goto` » qui irait *juste après* l'accolade fermante de la boucle ou du « `switch` » en question. Lorsque plusieurs boucles ou « `switch` » sont imbriqués les uns dans les autres, un « `break` » ne permet de sortir que de la boucle ou du « `switch` » le plus *interne*.

Voici un exemple d'emploi d'un « `break` » dans une boucle (la section suivante donnera des exemples de « `break` » dans des « `switch` ») :

```

int diviseur ;
for ( diviseur = 2 ; diviseur * diviseur <= nombre ; diviseur ++ ) {
    if ( nombre % diviseur == 0 )
        break;
}
if (diviseur * diviseur > nombre)
    printf("%d est un nombre premier\n", nombre);
else
    printf("Le plus petit diviseur de %d est %d\n", nombre, diviseur);

```

Le « `if (...) break;` » dans la boucle « `for` » compte pour *une* instruction. La boucle « `for` » n'a donc qu'une seule instruction et on a le droit d'enlever les accolades et d'écrire

```

for ( diviseur = 2 ; diviseur * diviseur <= nombre ; diviseur ++ )
    if ( nombre % diviseur == 0 ) break;

```

Deux remarques :

- Ce style d'écriture est assez acrobatique et peut être source d'erreurs et de confusion. Si on emploie ce style, il est indispensable d'avoir une indentation correcte.
- Le « `break` » fonctionne comme avant, même s'il n'y a plus d'accolade fermante. Le « `break` » est donc comme un « `goto` » après cette accolade fermante virtuelle qu'on aurait dû écrire si ces accolades n'avaient pas été optionnelles.

5.4 L'instruction « switch »

La syntaxe est la suivante :

```
switch ( expression ) {
    case valeur_1 :
        série d'instructions
    case valeur_2 :
        série d'instructions
    case valeur_3 :
        série d'instructions
    ... // etc. Il peut y avoir beaucoup de « case »

    default :
        série d'instructions
}
```

Ici, l'*expression* doit être de type entier (le plus souvent `int` ou `char`, mais pas exclusivement), et *valeur_1*, *valeur_2*, etc. doivent être des expressions *constantes* du même type.

L'*expression* est évaluée et comparée aux différentes constantes *valeur_1*, *valeur_2*, etc. Si une de ces constantes est égale au résultat de l'évaluation, un saut (comme avec un « `goto` ») est effectué au « `case` » correspondant. Si aucune constante ne correspond, un saut est effectué au label « `default` ».

Notez que c'est un deux-points et non un point-virgule qui suit chaque « `case valeur` » et le « `default` ». Ces lignes ne sont pas des instructions, mais marquent des endroits où le programme peut sauter. C'est cohérent avec les labels habituels où l'on peut sauter avec une instruction « `goto` » et qui sont également suivis de deux-points.

```
// On suppose que x, y et resultat sont des int et que c est un char qui contient le
// caractère correspondant à une opération arithmétique que l'on veut exécuter sur x et y.

switch( c ) {
    case '+' :
        resultat = x + y;
        break;
    case '-' :
        resultat = x - y;
        break;
    case '*' :
        resultat = x * y;
        break;
    case '/' :
        if ( y == 0 ) {
            printf("On ne peut pas diviser par zéro\n");
            printf("On abandonne\n");
            exit(1); // exit() arrête immédiatement le programme
        }
        resultat = x / y;
}
```

```

        break;
    default :
        printf("Impossible de comprendre l'opération « %c ».\n", c);
        printf("On abandonne\n");
        exit(1);
} // Chacun des « break » saute ici

printf ("Le résultat de %d %c %d est %d\n", x, c, y, resultat);

```

Dans l'exemple qui précède, la *série d'instructions* qui suit chaque « **case** » se termine par un « **break** » qui effectue un saut après l'accolade fermante du « **switch** ». Ces « **break** » sont ici indispensables ; en effet, après le saut initial, les labels (« *case valeur* » et « **default** ») sont ignorés et *toutes* les instructions qui suivent le point où l'on a sauté seraient exécutées s'il n'y avait pas les « **break** ». Dans la pratique, il faut presque toujours mettre un « **break** » à la fin de chaque *série d'instructions*.

Voici un exemple de « **switch** » où ce n'est pas le cas. On suppose que `get_an_answer()` est une fonction renvoyant un `int` qui lit et renvoie un nombre tapé au clavier par l'utilisateur :

```

printf("Tapez la Réponse : ");
switch ( get_an_answer() ) {
    case 42 :
        printf("Bravo, vous avez trouvé.\n");
        break;
    case 41 :
    case 43 :
        printf("Presque ! ");
    case 40 :
    case 44 :
        printf("Vous n'êtes pas loin de la Réponse.\n");
        break;
    default :
        printf("Non, ce n'est pas ça.\n");
}

```

Si l'utilisateur tape 41 ou 43, le « **switch** » saute à l'instruction « `printf("Presque ! ");` » et, au final, ce fragment de programme affiche « Presque! Vous n'êtes pas loin de la Réponse. ».

Dans un « **switch** », il est interdit d'avoir deux labels « *case valeur* » avec la même valeur. De même, il est interdit d'avoir deux labels « **default** ».

On peut écrire un « **switch** » sans label « **default** ». Dans ce cas, si aucun des labels « *case valeur* » n'a la bonne valeur, aucune des instructions entre les accolades du « **switch** » n'est exécutée.

Chapitre 6

Entrées-sorties

6.1 Afficher avec printf

`printf(format, exp1, exp2, ...)` permet d'afficher sur la « sortie standard », c'est-à-dire en général sur l'écran, les valeurs des expressions `exp1`, `exp2`, ... en utilisant le format d'affichage `format`. C'est une fonction très particulière, puisque le nombre et le type des arguments sont différents d'un appel à l'autre. C'est, entre autres, le rôle de l'argument `format` de spécifier le nombre et le type des arguments qui suivent. Par exemple

```
int nboui, nbttotal;
[...] ; // On donne des valeurs à nboui et nbttotal
printf("%d personnes sur %d ont répondu « Oui », soit %g%%.\\n",
      nboui, nbttotal, 100.0*nboui/nbttotal);
```

Le paramètre `format` est une chaîne de caractères qui sera affichée après interprétation des codes spéciaux, qui commencent par les caractères « % ». Les deux premiers codes (deux fois « %d ») sont remplacés par des entiers (écrits en base dix) `nboui` et `nbttotal`. Le code « %g » est remplacé par le nombre à virgule `100.0*nboui/nbttotal`. Enfin, le code « %% » est remplacé par le caractère « % ».

Il est important de bien ajuster les codes dans le format avec le nombre et le type des autres arguments. Dans l'exemple, il y a quatre codes qui ont besoin, dans cet ordre, de deux entiers et d'un nombre à virgule, et il y a bien dans l'appel à `printf` trois arguments, en plus du format, et qui sont des types demandés. Notez en particulier que le C est incapable de faire tout seul les conversions nécessaires puisque les types des arguments sont inconnus :

```
double x;
x= sqrt(48.0); // Correct: sqrt prend un double et retourne un double.
x= sqrt(48); // Correct aussi: le C sait que sqrt prend un double, et il va convertir
// le 48 en 48.0 avant d'appeler la fonction.

printf("%g\\n", 48.0); // Correct: le %g demande un double et 48.0 en est un.
printf("%g\\n", 48); // INCORRECT: le %g demande un double et 48 est un entier. Comme
// le type des arguments de printf est variable, le C ne sait pas
// qu'il faut convertir le 48 en 48.0
```

L'affichage que fait `printf` n'est pas toujours celui désiré. Par exemple,

```
|| printf("%d personnes sur %d font %g%%. \n", 387, 916, 100.0*387/916);
```

affiche « 387 personnes sur 916 font 42.2489%. ». Pour les résultats d'un sondage, trop de chiffres après la virgule sont affichés ! On peut demander à `printf` de n'afficher que trois chiffres significatifs de la manière suivante :

```
|| printf("%d personnes sur %d font %.3g%%. \n", 387, 916, 100.0*387/916);
```

ce qui affiche « 387 personnes sur 916 font 42.2%. ». De manière générale, un code de `printf` a la forme suivante :

`%<flags><taille><précision><taille du type><conversion>`

`<conversion>` est le seul paramètre obligatoire ; il indique ce qu'on doit afficher. Les exemples qu'on a déjà vus sont les lettres « d », « g », « c », « s » qui indiquent respectivement, un entier, un nombre à virgule, un caractère ou une chaîne de caractères.

`<taille du type>` sert à indiquer précisément le type de l'argument. Les cas les plus courants sont « l » pour indiquer que l'argument est un entier de type `long`, « ll » pour indiquer que c'est un entier de type `long long`, et « L » pour indiquer que c'est un nombre à virgule de type `long double`.

`<précision>` a la forme d'un point suivi d'un entier. Il sert essentiellement à donner le nombre de chiffres significatifs avec lequel on doit afficher un nombre à virgule. (Voir l'exemple plus haut avec « `%.3g` ».)

`<taille>` est un nombre entier qui indique le nombre minimal de caractères que doit comporter l'affichage : si la conversion utilise moins de caractères que demandés, elle est précédée du bon nombre de caractères « *espace* » pour compléter. (La conversion est donc calée à droite.)

`<flags>` représente zéro, un ou plusieurs caractères (appelés *flags*) parmi « - », « + », « *espace* », « # » ou « 0 » (zéro) qui modifient comment la valeur est affichée. Le *flag* « - » signifie que si une `<taille>` est indiquée et que la conversion utilise moins de caractères que demandées, on doit la compléter en la faisant suivre de caractères « *espace* ». (La conversion est donc calée à gauche plutôt qu'à droite.) Par exemple :

```
|| printf(" [%s] [%8s] [%-8s] \n","un","deux","trois");
```

affiche « [un] [____deux] [trois____] ».

Les autres *flags* n'ont de sens que pour l'affichage des nombres. Le *flag* « 0 » signifie que si une `<taille>` est indiquée et que la conversion utilise moins de caractères que demandées, on doit la compléter en la faisant précéder de caractères « 0 » plutôt que d'espaces. Par exemple,

```
|| printf(" [%d] [%4d] [%-4d] [%04d] \n", 42, 42, 42, 42);
```

affiche « [42] [__42] [42__] [0042] ».

Le *flag* « + » signifie que le nombre doit toujours être précédé d'un signe moins ou plus. (Par défaut, seuls les signes moins sont affichés.) Le *flag* « *espace* » signifie que le nombre doit toujours être précédé soit d'un signe moins, soit d'un espace. Par exemple,

```
|| printf("[%g] [%g] [%+g] [%+g] [% g] [% g]\n",1.5, -2.3, 1.5, -2.3, 1.5, -2.3);
```

affiche « [1.5] [-2.3] [+1.5] [-2.3] [_1.5] [-2.3] »

Le sens du *flag* « # » dépend du type de conversion demandé, voir plus loin.

Entrons maintenant dans les détails.

6.1.1 Afficher des entiers avec printf

La lettre « d » ou « i » ont le même sens et permettent d’afficher un entier signé (de type `int`). Pour afficher un entier non signé (de type `unsigned int`), il faut utiliser la lettre « u ».

On peut aussi utiliser les lettres « o », « x » et « X » pour afficher un entier non signé en octal, en hexadécimal en utilisant les lettres « a, b, c, d, e, f » ou en hexadécimal en utilisant les lettres « A, B, C, D, E, F ».

Ces lettres doivent être précédées de « l » pour des entiers de type `long` et `unsigned long`, et de « ll » pour des entiers de type `long long` et `unsigned long long`.

On peut aussi les faire précéder de « h » pour des entiers de type `short` et `unsigned short`, et de « hh » pour des entiers de type `signed char` et `unsigned char`. C’est cependant habituellement inutile vu les règles de conversions du C. Il existe aussi les modificateurs « j », « z » et « t » qui indiquent respectivement que l’argument est de type `intmax_t` (ou `uintmax_t`), `size_t` ou `ptrdiff_t`.

Le paramètre *<précision>* indique le nombre minimal de chiffres à afficher. Par exemple,

```
|| printf(" [%- 8.4d] \n", 42);
```

affiche « [_0042_] » : deux zéros ont été rajoutés pour afficher quatre chiffres, le tout tient dans un espace de huit caractères, calé à gauche, mais avec quand même un espace à gauche représentant la place qu’aurait pris le signe moins.

Si la précision vaut 0 et que l’entier à afficher vaut 0, rien n’est affiché. Le *flag* « # » ne fonctionne qu’avec les conversions « x », « X » et « o ». Pour « x » et « X », un résultat non nul est préfixé de « 0x » ou « 0X », et pour « o », la précision est ajustée si nécessaire pour que le nombre commence toujours par un 0.

6.1.2 Afficher des nombres à virgule avec printf

Les lettres « e », « E », « f », « F », « g » et « G » permettent d’afficher des nombres à virgules. Ces lettres doivent être précédées de « L » si l’on désire afficher la valeur d’un nombre de type `long double`. Les lettres « e » et « E » affichent les nombres en notation scientifique sous la forme `6.02e+23` (pour « e ») ou `6.02E+23` (pour « E »). Les lettres « f » et « F » affichent les nombres sous forme décimale, et « g » et « G » choisissent selon le nombre une forme ou l’autre. Par exemple,

```
|| printf("%e %f %g ; %e %f %g\n",3.14, 3.14, 3.14, 299792458., 299792458., 299792458.);
```

affiche « 3.140000e+00 3.140000 3.14 ; 2.997925e+08 299792458.000000 2.99792e+08 ». Pour les lettres « e », « E », « f » et « F », la *<précision>* est le nombre de chiffres après la virgule et vaut par défaut 6. Pour les lettres « g » et « G », la *<précision>* est le nombre de chiffres significatifs et vaut par défaut 6. Le nombre sera affiché sous forme décimale s’il est compris en valeur absolue entre 10^{-4} (inclus) et $10^{<précision>}$ (exclus), et sous forme scientifique sinon (avec un e ou un E selon que la lettre est « g » ou « G ».)

Toutes ces lettres peuvent aussi afficher des nombres à virgules spéciaux : `printf("%g %G\n", 1./-0., 0./0.)` ; affiche, par exemple, « -inf NAN » ou quelque chose d’approchant. « -inf » signifie bien sûr $-\infty$, et « NAN » est

l'abréviation de *not a number*. Le premier est en minuscule et le deuxième en majuscule parce qu'on a utilisé « g » et « G », respectivement.

Normalement, le point décimal n'est pas affiché si aucun chiffre ne le suit, et si les derniers chiffres significatifs d'un nombre sont des zéros, « g » ou « G » ne les affichent pas : `printf("%.0f %g\n", 123.0, 123.0)`; donne « 123 123 ». Si on met le *flag* « # », ces simplifications disparaissent : `printf("%#.0f %#g\n", 123.0, 123.0)`; donne « 123. 123.000 »

Les lettres « a » et « A » permettent d'afficher un nombre à virgule en notation scientifique hexadécimale. La précision par défaut est telle que le nombre est exactement représenté. Par exemple, `printf("%a\n", 1234.5)`; donne « 0x1.34ap+10 » ce qui signifie $1 \times 2^{10} + 3 \times 2^6 + 4 \times 2^2 + 10 \times 2^{-2}$.

6.1.3 Afficher des caractères avec printf

La lettre « c » permet d'afficher un caractère :

```
|| printf("%c\n", 'A');
```

Rappelons qu'en C, un caractère est un nombre entier. Ainsi,

```
|| printf("%c\n", 65);
```

est valide et affiche « A », car 65 est le code de la « A » dans le codage habituel, et

```
|| printf("%d\n", 'A');
```

affiche « 65 ».

6.1.4 Afficher des chaînes de caractères avec printf

La lettre « s » permet d'afficher une chaîne de caractères :

```
|| printf("%s\n", "Bonjour tout le monde");
```

On peut indiquer une *<précision>* qui est le nombre maximal de caractères affichés. Par exemple,

```
|| printf("%.10s\n", "Bonjour tout le monde");
```

affiche « Bonjour to ».

6.1.5 Remarques diverses sur printf

« % » a un sens particulier dans la chaîne de format. Si on veut effectivement afficher ce caractère, il faut le doubler et taper « %% ».

Si on veut afficher une chaîne de caractère contenu dans un tableau `x` (ou repéré par un pointeur), il faut absolument faire

```
|| printf("%s", x);
```

et non `printf(x)`; Cette dernière forme fera en effet des erreurs bizarres le jour où la chaîne `x` contiendra le caractère « % ».

`printf` renvoie un entier égal au nombre de caractères affichés.

Au lieu de donner des valeurs aux paramètres *<taille>* et *<précision>* dans le format, on peut mettre le caractère « * » dans le format et donner la valeur numérique sous la forme d'un argument (de type entier) supplémentaire. Par exemple, `printf("%. *g\n", pres, x)`; affiche le nombre à virgule « x » avec un nombre de chiffres significatifs qui est contenu dans l'entier « pres ».

Le code « %n » permet de récupérer le nombre de caractères qui ont été écrits jusque là. L'argument correspondant doit être un pointeur sur un entier où ce nombre de caractères sera écrit. Par exemple,

```
|| int x; printf("%g;%n;%d\n",123.0, &x, 42);
```

affichera « 123;42 », et la variable `x` contiendra après l'appel de fonction le nombre 4.
Le code « `%p` » permet d'afficher la valeur d'un pointeur sur `void`.