Le langage C

Introduction, guide de référence

Ce document est une présentation du langage de programmation C, de sa syntaxe et de ses spécificités. Il est destiné essentiellement à servir de mémo-guide de référence syntaxique.

Ce guide n'est PAS un manuel de programmation, et s'adresse de fait à un lectorat ayant déjà une pratique raisonnable d'un autre langage informatique évolué (typiquement Pascal ou Fortran-77) et désirant ou ayant besoin d'utiliser C.

Enfin, il est bon de signaler que l'environnement de programmation C disponible sur plateformes Unix (librairies système et utilitaires) étant quelque chose de très volumineux, ce guide présente et commente un certain nombre de fonctions d'usage courant, mais ne peut prétendre couvrir de manière exhaustive les centaines d'appels disponibles!

Édition 2.0

Auteur Jean-François Rabasse

Contact :
jean-francois.rabasse@lra.ens.fr
http://www.lra.ens.fr/

Sommaire

1 Présentation

Le langage C a été créé au milieu des années 1970, aux Bell Laboratories, par Brian Kernighan et Denis Ritchie, initialement pour écrire le système d'exploitation Unix. La première version a été rendue disponible vers 1977, la référence syntaxique étant alors l'ouvrage informel publié par Kernighan et Ritchie, *The C Programming Language*.

Cette version reste aujourd'hui désignée sous le nom de "**C-K&R**".

La syntaxe a évolué dans les années 1980, donnant lieu à une multitude de variations, extensions constructeurs, spécificités machines, etc., pour aboutir à une normalisation par l'Ansi. L'appellation "**C-Ansi**", ou "**C-Ansi** 89", désigne la syntaxe officialisée par la norme X3.159-1989 de février 1990.

Outre des améliorations sensibles dans la rigueur d'écriture des programmes (contrôles de types, contrôles d'appels), le **C-Ansi** offre une garantie de portabilité des développements effectués sur tous les environnements disposant d'un compilateur aux normes.

Il n'existe aujourd'hui aucune bonne (ou mauvaise) raison de ne pas adopter la syntaxe Ansi !

1.1 Mise en oeuvre

Par convention, les fichiers sources C sont désignés par un nom comportant une extension `.c`

Pour des raisons de compatibilité ascendante, les compilateurs modernes supportent, par défaut, l'ancienne syntaxe du **C-K&R**. La compilation explicite en **C-Ansi** se fait via une option de compilation.

Encore une fois, et sauf raison majeure, il faut utiliser la norme syntaxique de l'Ansi qui apporte une réelle sécurité dans l'écriture des codes C.

Les commandes de compilation (et options) les plus courantes sont :

- DEC C sur machines VAX/VMS

```
cc/standard=ansi89 toto.c
```

- DEC C sur machines Alpha/DEC-Unix

```
cc -c -std1 toto.c
```

- Sun C sur machines Sun/Solaris

```
cc -c -Xc toto.c
```

- GNU C sur toutes machines

```
gcc -c -ansi toto.c
```

L'édition de liens du ou des modules objets se fait de manière classique :

- Linker VMS

```
link/exec=toto.exe toto.obj
```

- Linkers Unix

```
cc -o toto toto.o
```

ou, en cas d'utilisation du GNU C,

```
gcc -o toto toto.o
```

1.2 Utilisation de librairies

Comme pour d'autres langages, l'utilisation de librairies externes depuis du code C passe par :

1. Déclarations des constantes et fonctions de la librairie désirée par inclusion dans le programme source d'un fichier en-tête :

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

2. Édition de liens avec la ou les librairies désirées :

```
cc -o toto toto.o -lm
```

NB : en C, les inclusions de fichiers se font par une directive, **`#include`**, dont la mise en oeuvre est détaillée en annexe A page 45.

2 Structure du langage

Comme tous les langages postérieurs à 1969, C est un langage "pascalien" : langage modulaire, à présentation libre, à déclarations préalables obligatoires[1] et à variables locales gérées par contextes autorisant les programmations récursives.

De plus, la compilation d'un programme source C s'effectue en deux phases successives :

1. Le *préprocesseur* effectue une mise en forme lexicale, élimination des espaces, lignes vides et commentaires, inclusions de fichiers externes, définitions de constantes symboliques et macro instructions, gestion de code conditionnel, ...
2. Le compilateur proprement dit traite la sortie du préprocesseur et génère le fichier objet.

L'utilisation du préprocesseur se fait en incorporant dans le texte source des *directives*, qui sont des mots-clés précédés du caractère **#**, par exemple **#include**, **#define**, **#if**, etc. Se reporter à l'annexe A page 45 pour les détails de mise en oeuvre. Voir également la note [2.1] page 8 .

2.1 Éléments lexicaux

2.1.1 Présentation

C, comme Pascal, fait partie des langages dits "à présentation libre", i.e. les passages à la ligne dans l'écriture des sources ne sont pas significatifs (contrairement à Fortran). Une fin d'instruction doit être marquée explicitement par le caractère **;**

Ainsi les écritures :

```
x = 44.5; y = 2 * x - 0.5; z = x / y;
```

ou :

```
    x = 44.5;
y = 2 * x - 0.5;
    z = x
      /
      y       ;
```

sont strictement équivalentes.

Enfin, les séparateurs, espaces ou tabulations, sont facultatifs et peuvent figurer en nombre quelconque.

(1) Le mécanisme de déclaration implicite de Fortran n'existe pas, ce qui est une excellente chose !

2.1.2 Commentaires

C utilise des commentaires "par blocs", et non orientés ligne comme en Fortran (voir note [2.2] page 8).

Les blocs commentaires sont délimités par les séquences de caractères **/*** et ***/** :

```
/*
   EXEMPLE LEXICAL
   (Mise a jour septembre 96)
*/
```

```
float x;    /* Coordonnee x */
float y;    /* Coordonnee y */
```

2.1.3 Mots-clés

Les mots-clés du langage sont strictement réservés et ne peuvent être utilisés comme identificateurs. (En tout état de cause, l'intérêt de pouvoir nommer des variables **if** ou **goto**, comme Fortran l'autorise, ne semble pas évident !)

2.1.4 Casse caractères

Contrairement à Pascal et Fortran, C distingue majuscules et minuscules dans les identificateurs et les mots-clés du langage :

```
float toto, Toto, totO;  /* Trois variables distinctes */
Float y;  /* Erreur en compilation, Float est inconnu  */
```

Tous les mots-clés du C doivent être saisis en minuscules, les identificateurs d'un programme sont définis, eux, au gré du programmeur.

NB : lorsque l'on utilise des macro définitions du préprocesseur (voir A.2 page 45), il est habituel d'utiliser des identificateurs majuscules. Ceci n'est qu'une convention destinée à attirer l'attention lors de la lecture des codes sources C.

NB : bien que les conventions lexicales du langage le permettent, il est fortement déconseillé d'utiliser des noms symboliques ne se distinguant que par la casse des caractères. Sur certains systèmes (VMS, MS-DOS), les éditeurs de liens, eux, ne distinguent pas les majuscules des minuscules dans les fichiers objets. On risque donc des collisions d'identificateurs !

2.1.5 Identificateurs

Les identificateurs, de variables, de fonctions, obéissent à la règle habituelle "suite de caractères alphanumériques, l'initiale devant être une lettre". Le caractère de soulignement _ est autorisé. L'utilisation d'autres caractères non alphanumériques est à éviter dans la mesure du possible (voir note [2.3] page 8).

La taille maximale d'un identificateur était limitée à 6 caractères en **C-K&R**. Aujourd'hui tous les compilateurs Ansi acceptent un minimum de 31 caractères[2]. L'utilisation de noms assez longs a l'immense avantage de mieux documenter un

(2) Et souvent beaucoup plus, 128 est une valeur courante.

programme :

```
float constante_de_planck;
```

est probablement plus explicite que :

```
float h;
```

NB : il faut éviter d'utiliser le caractère de soulignement _ en début ou fin d'un identificateur, cette notation est habituellement réservée aux mécanismes internes des compilateurs.

2.2 Structure source

Par rapport à un langage comme Fortran, pour lequel les règles de visibilité des identificateurs sont très rigides (une déclaration ne vaut que dans le module de code, **subroutine** ou **function**, où elle est déclarée), C offre une plus grande souplesse, et utilise des notions de *classe* de déclaration et de *durée de vie* des variables.

L'exemple suivant (un peu artificiel) va permettre d'illustrer ces différents mécanismes et d'introduire la terminologie associée :

```
/*
    EXEMPLE SOURCE C
    ----------------
*/
#include <math.h>       /* Globale          */
#define PI  3.141592    /* Globale, privee  */
float param_a;          /* Globale, publique, statique  */
static float param_b;   /* Globale, privee, statique    */
extern float coeff;     /* Globale, externe, statique   */
static void setup(float a)  /* Globale, privee      */
{
    param_a = a;
    param_b = a / 3.5;
}
float linear1(float x)      /* Globale, publique    */
{
    return coeff * (param_a * x + param_b);
}
float linear2(float x, float param_b)
{
    static float param_a = 2.5; /* Locale, statique */
    auto float multi;       /* Locale, automatique  */
    multi = coeff * 1.05;
    return multi * (param_a * x + param_b);
}
float linear3(float x)
{
    return coeff * linear1(x);
}
```

2.2.1 Déclarations globales

Un fichier source C peut comporter une ou plusieurs définitions de modules fonctions, quatre dans l'exemple ci-dessus. Une définition de fonction est constituée d'une en-tête (interface) et d'un *corps* de fonction délimité par une paire d'accolades.

➤ On appelle *déclaration globale* toute déclaration effectuée à l'extérieur des corps de fonctions, donc au "niveau zéro" du texte source. C'est le cas des trois variables **`param_a`**, **`param_b`** et **`coeff`** de l'exemple. Un identificateur global est valide (ou encore "visible") pour tous les modules fonctions d'un même fichier source[3].

Par voie de conséquence, les déclarations issues d'une inclusion de fichier en tête d'un source sont globales, ainsi que les constantes symboliques définies (symbole **`PI`** de l'exemple). De même, la définition d'une fonction a valeur de déclaration globale. Ainsi, la fonction **`linear3`** de l'exemple utilise (ou "appelle") la fonction **`linear1`** sans spécification ou typage particulier.

➤ On appelle *déclaration locale*, toute déclaration effectuée dans un corps de fonction. C'est le cas des deux variables **`param_a`** et **`multi`** de la fonction **`linear2`**. Un identificateur local n'est utilisable que dans le corps de fonction auquel il appartient.

NB : les déclarations des arguments d'appel d'une fonction sont des identificateurs locaux.

NB : en cas d'homonymie, c'est la déclaration locale qui est prioritaire. Dans la fonction **`linear2`** de l'exemple, les variables locales **`param_b`** (argument d'appel) et **`param_a`** "masquent" les variables globales de mêmes noms du fichier source.

2.2.2 Variables statiques

Ce mécanisme, à ne pas confondre avec le précédent, concerne la durée de vie des variables. C est un langage à *contextes d'exécution* c'est à dire que, lors d'un appel de fonction, une zone mémoire est allouée dans la pile d'exécution du processeur, destinée à loger les variables nécessaires.

➤ Une *variable automatique* est une variable créée dans le contexte d'exécution d'une fonction, au moment de l'appel. C'est le cas de la variable **`multi`** de la fonction **`linear2`** de l'exemple. Un contexte d'exécution étant détruit lorsque l'on quitte la fonction, les variables automatiques disparaissent. Entre autres conséquences, cela signifie qu'en C, lors d'appels successifs à une même fonction, on ne retrouve jamais les variables locales avec la valeur qu'elles avaient lors de l'appel précédent !

➤ Une *variable statique* est une variable à laquelle est assigné un emplacement mémoire fixe, et dont la durée de vie est celle du programme en cours d'exécution (contexte de programme par opposition à un contexte de fonction).

Particularités :

- Une déclaration globale (effectuée en dehors d'un corps de fonction) crée nécessairement une variable statique, puisque non associée à un contexte particulier autre que celui du programme.
- Par défaut, toute déclaration locale crée une variable automatique. Le spécificateur **`auto`**, utilisé dans l'exemple pour déclarer la variable **`multi`** de la fonction **`linear2`**, est superflu. On peut même dire qu'il ne sert à rien en C et on ne le rencontre pratiquement jamais dans des codes !

(3) Grosse différence par rapport à Fortran, on n'a pas à redéclarer les mêmes choses dans chaque module.

- Le contexte d'exécution d'une fonction est alloué au moment de l'appel. C permet donc les programmations récursives puisque chaque appel ou réappel va disposer de son propre jeu de variables.
- La création d'une variable locale (à une fonction) et statique (durée de vie hors contexte) nécessite l'utilisation du spécificateur **`static`**. C'est le cas pour la variable **`param_a`** de la fonction **`linear2`** de l'exemple.

  Une telle variable conserve donc sa valeur d'un appel à l'autre de la fonction[4]. Par contre, on perd toute possibilité de récursion.

2.2.3 Déclarations publiques, privées

Ce dernier mécanisme concerne la visibilité et l'accessibilité entre différents modules sources et objets d'un même programme.

➤ On appelle *identificateur public* un identificateur de variable ou fonction exploitable depuis un autre module source du programme. Les références publiques seront traitées lors de l'édition des liens entre les divers modules objets d'une application.

➤ On appelle *identificateur privé* un identificateur de variable ou fonction réservé au module qui le déclare, et inutilisable depuis un autre module du programme. Concrètement, cela signifie que le compilateur ne placera pas l'identificateur en question dans la table des symboles du fichier objet. Un appel depuis un autre module ne sera pas résolu au moment de l'édition de liens.

Par défaut, tous les identificateurs de variables ou fonctions globales d'un fichier source sont publics. Lorsqu'on désire créer une variable ou fonction privée, on utilisera le spécificateur **`static`**. C'est le cas de la variable **`param_b`** et de la fonction **`setup`** de l'exemple.

Attention : une incohérence syntaxique du C fait que le même spécificateur **`static`** est utilisé pour deux choses sans aucun rapport entre elles !

- **`static`** appliqué à une variable locale à une fonction spécifie une durée de vie, *statique* par opposition à *automatique* qui est le défaut.
- **`static`** appliqué à une fonction ou à une variable globale (qui est donc nécessairement *statique* au sens de la durée de vie) indique une déclaration *privée* par opposition à *publique* qui est le défaut.

Même s'il n'y a jamais ambiguïté, pour le compilateur, cette confusion est fâcheuse pour l'humain. L'introduction dans le langage d'un mot-clé dédié, par exemple "**`private`**" ou autre, aurait été préférable. Dommage...

➤ Enfin, on appelle *identificateur externe* l'identificateur d'une variable appartenant à un autre module source, nécessairement globale et publique. Le spécificateur **`extern`** a pour fonction d'informer le compilateur de l'invocation et du type de cette variable. La résolution se fera lors de l'édition de liens.

NB : lorsqu'une variable, globale et publique, est utilisée par plusieurs modules objets d'une même application, elle doit être définie dans un et un seul des modules sources, et déclarée **`extern`** dans tous les autres !

(4) C'est ce type de variable qu'implante un langage comme Fortran.

2.3 Notes commentaires

[2.1] Il est utile de signaler que le préprocesseur C est un processeur de texte, pas obligatoirement lié au langage lui-même.

Certains compilateurs Fortran modernes l'utilisent également pour mettre en forme un texte source (Fortran) avant d'appeler le compilateur proprement dit. Ceci permet, même depuis un développement en Fortran, d'exploiter les puissantes fonctions de compilation conditionnelle et de macro substitution du préprocesseur C.

En pratique, on écrira un source Fortran, dans un fichier dont l'extension sera **.F** au lieu du **.f** habituel, qui comportera des directives du préprocesseur. La compilation :

```
f77 toto.F
```

s'effectuera en deux temps, comme pour du C : prétraitement puis appel du compilateur Fortran avec le résultat.

[2.2] Le langage C++, dérivé de C, utilise les commentaires **/*** et ***/** de C et également un commentaire de type "jusqu'à fin de ligne" sous la forme :

```
// Commentaire
```

Certains compilateurs C, compatibles C++, acceptent de fait la notation **//**.

Cette utilisation est déconseillée dans la mesure où elle n'apporte rien de fondamental et n'assure plus la portabilité du code ainsi écrit !

[2.3] Certains environnements ont une tradition lexicale parfois incompatible avec les règles du **C-Ansi**. Par exemple, le caractère **$** cher à Digital n'est, en principe, pas admis dans des identificateurs C.

Les compilateurs disponibles sur ces environnements auront des options permettant d'étendre le standard. Avec le DEC C, sous VMS, l'option :

```
cc/standard=relaxed_ansi89 toto.c
```

permet d'accepter des déclarations du style :

```
float constante$de$planck;
```

Sauf dans les cas où l'on ne peut faire autrement (programme en C, sous VMS, appellant des routines système, **lib$get_truc** et autres **sys$machin**), il est préférable d'éviter ces exotismes et de s'en tenir à la stricte syntaxe portable.

3 Déclarations, types, prototypes

C, comme Pascal, est un langage "à déclarations préalables", i.e. toutes les variables utilisées par un module doivent avoir été déclarées et typées. Les mécanismes de déclaration implicite du Fortran n'existent pas.

De plus, en **C-Ansi**, toutes les fonctions appelées doivent être préalablement déclarées et prototypées. Le prototype est une description de l'interface d'appel de la fonction (nombre d'arguments et nature de ceux-ci). Ce mécanisme permet au compilateur de faire tous les contrôles nécessaires, et les éventuelles conversions de types.

Enfin, comme on l'a vu au 2.2 page 5, une déclaration C peut également comporter des spécifications de visibilité ou de durée de vie. La syntaxe générale d'une déclaration C est :

```
<classe> <attribut> <type> <nom> ;
```

où **<type>** et **<nom>** sont obligatoires et **<classe>** et **<attribut>** optionnels.

3.1 Types scalaires de base

C dispose, en standard, des types scalaires suivants :

char Numérique entier sur 8 bits (idem **byte** de Fortran). Gamme de valeurs de -128 à +127.

short Entier "court" sur 16 bits (idem **integer*2** de Fortran). Gamme de valeurs de -32768 à +32767.

int Entier par défaut machine (idem **integer** de Fortran). La taille dépend des plateformes. Sur la plupart des machines actuelles, VAX, Sun, DEC Alpha, l'**int** C est un mot de 32 bits.

long Entier "long". Sa taille est au minimum 32 bits sur toutes machines (même sur des machines 16 bits type PC sous MS-DOS). Il peut être plus long. Le **long** C fait 32 bits sur des VAX, des Sun, des PC sous Linux et 64 bits sur des DEC Alpha.

float Réel simple précision (idem **real*4** de Fortran).

double Réel double précision (idem **real*8** de Fortran).

void Pseudo type "rien", introduit avec le **C-Ansi** pour des raisons de consistance syntaxique. Ce déclarateur est décrit au paragraphe 3.9 page 16.

NB : comme son nom ne l'indique pas, le type **char** n'est PAS un type caractère, mais un type numérique entier. Contrairement à d'autres langages (Pascal, Fortran), C ne dispose pas de type spécifique caractère et confond le caractère au sens lexical avec son code numérique Ascii. Le **char** est donc le plus petit entier permettant de manipuler des codes caractères (voir note [3.1] page 18).

Cette confusion fait qu'on ne trouvera donc pas, en C, de fonctions de conversion caractère en numérique et inverse analogues à **chr()** et **ord()** de Pascal ou **char()**

et **`ichar()`** de Fortran.

NB : C ne dispose pas non plus de type spécifique booléen, contrairement à Pascal (**`boolean`**) et à Fortran (**`logical`**). La notion de booléen est supportée par des variables ou expressions arithmétiques, avec la convention "expression nulle ou non nulle" pour traduire "expression fausse ou vraie". Le paragraphe 4.4 page 23 développera ces notions.

3.2 Extensions

Certaines implantations de C proposent d'autres types scalaires :

long long
: Entier "très long", typiquement 64 bits. Ce type est disponible dans les environnements C destinés à des plateformes 64 bits, et où le **`long`** standard est implanté en entier 32 bits. On trouvera ce type sous certains compilateurs (GNU C) pour des machines Sun ou Linux. Il est inutile sur des machines DEC-Alpha où le **`long`** est déjà implanté en 64 bits.

long double
: Type réel étendu par rapport au réel **`double`** 64 bits. Ce type, lorsqu'il est disponible, permet d'utiliser un format flottant propre aux unités de calcul arithmétique (FPU), en général sur 80 bits.

  L'utilisation de ce type est discutable[5]. Les FPU travaillent habituellement sur 80 bits pour assurer la précision de calcul en 64 bits en éliminant les erreurs d'arrondi. Travailler explicitement en 80 bits peut donc donner une fausse illusion de précision augmentée !

L'utilisation de ces extensions, non standard, doit se faire en cas de besoin avéré, la portabilité n'étant plus du tout garantie.

3.3 Attributs de types

En complément des déclarateurs de type, qui spécifient la nature de la donnée, C dispose de qualifieurs complémentaires qui doivent être définis devant la déclaration de type.

3.3.1 Entiers non signés

Le qualifieur **`unsigned`**, applicable uniquement aux types entiers, signale une interprétation non signée d'une donnée. Cet attribut permet d'adapter la gamme des valeurs utiles une fois la taille choisie.

(5) Affirmation parfaitement subjective et qui n'engage que moi !

Le choix de l'une ou l'autre représentation dépend de l'usage prévu pour la donnée.

```
char c1;             /* Utilisable de -128 a +127     */
unsigned char c2;    /* Utilisable de 0 a 255         */
short s1;            /* Utilisable de -32768 a +32767 */
unsigned short s2;   /* Utilisable de 0 a 65535       */
etc.
```

NB : lorsque le type de l'entier est un **int**, type par défaut en C, l'attribut **unsigned** peut être utilisé seul :

```
unsigned toto;
```

déclare un **unsigned int** implicitement.

NB : il existe également un qualifieur **signed**, parfaitement inutile puisque c'est la représentation par défaut des scalaires entiers.

3.3.2 Données constantes

Le qualifieur **const**[6] permet de déclarer une donnée non modifiable dans un contexte particulier.

L'intérêt est d'abord d'apporter une rigueur d'écriture et une sécurité syntaxique, le compilateur protestera en cas de tentative d'affectation :

```
int toto1 = 1;
const int toto2 = 2;
toto1 = 10;   /* Licite                 */
toto2 = 12;   /* Erreur en compilation */
```

D'autre part, sur certaines plateformes, le compilateur pourra exploiter des fonctionnalités système et ranger des données globales **const** dans des segments mémoire *read-only*.

3.3.3 Données fugitives

Le qualifieur **volatile**[6] permet de déclarer une donnée *fugitive*. Par donnée fugitive, on entend une variable susceptible d'être modifiée même si le code environnant n'effectue pas une affectation explicite. Le cas typique est la mise a jour d'une donnée de manière asynchrone, depuis un *handler* d'interruptions ou une routine *callback*.

Cette précaution sert à empêcher un compilateur optimiseur de faire des bêtises en croyant que telle ou telle variable est immuable. Ainsi, le fragment de code suivant :

```
int laurel;
int hardy;
laurel = 10;
while( laurel > hardy ) {
    ... Faire quelque chose
}
```

dans lequel la variable *laurel* est modifiée de manière externe et non explicitement dans la boucle, mettra en défaut un bon compilateur optimiseur.

Celui-ci, croyant bien faire, adaptera le code pour effectuer un test de boucle sur

(6) Introduit avec le **C-Ansi**, non disponible en **C-K&R**.

`10 > hardy`, sans "relire" la variable *laurel* à chaque itération.

La solution passe par une déclaration :

```
volatile int laurel;
```

L'alternative consisterait à ne pas compiler en mode optimisé, ou à utiliser un mauvais compilateur optimiseur[7].

3.4 Classes d'affectation

Ces spécificateurs, sauf un, ont déjà été évoqués au paragraphe 2.2 page 5. On les rappelle :

auto Déclare une variable locale *automatique*. Spécificateur jamais utilisé puisque c'est le défaut pour toute variable locale.

static Appliqué à une variable locale, ce spécificateur demande un stockage statique, la durée de vie de la variable est celle du programme et non celle du contexte local d'exécution.

Appliqué à une variable globale ou à une fonction, ce spécificateur déclare un identificateur privé au module source.

extern Déclare une variable, et son type, appartenant à un autre module source de l'application.

register Ce spécificateur très particulier demande un stockage de la variable associée dans un registre du CPU, et non en mémoire. Il n'est applicable qu'à des types scalaires de taille ad-hoc (compatible avec la taille des mots machine) et à des variables locales.

Son intérêt est un accès à la variable extrêmement efficace. L'utilisation typique est la déclaration de variables temporaires de type "compteurs de boucles" par exemple.

```
register int jj, ii;
for(ii = 0; ii < 10000; ii = ii + 1) {
    for(jj = 0; jj < 10000; jj = jj + 1) {
    ...
    }
}
```

NB : contrairement aux autres déclarateurs, la classe **`register`** n'est pas impérative. Il s'agit plutôt d'une suggestion, le compilateur fera au mieux. Ainsi rien n'interdit de déclarer une quarantaine de variables **`register`** même sur un CPU qui ne comporte que 24 ou 32 registres. Le compilateur allouera selon ses possibilités et toujours dans l'ordre des déclarations. Ainsi, dans l'exemple ci-dessus, si le compilateur ne peut affecter qu'un seul registre, c'est la variable **`jj`** qui en bénéficiera, **`ii`** sera une variable automatique normale.

C'est pourquoi on a toujours avantage à déclarer des variables registres avec une logique de priorité. Dans le cas de boucles imbriquées, l'efficacité porte d'abord sur la boucle interne. On déclare donc, dans l'ordre, **`jj`** puis **`ii`**.

(7) Qu'on ne compte pas sur moi pour citer des noms, je ne suis pas un délateur !

3.5 Alias de types

C dispose d'un outil très utile permettant de définir de nouveaux types de données à partir de types existants, par l'instruction :

```
typedef <type> <alias>;
```

- On utilise couramment l'alias de type pour alléger des notations :

```
typedef unsigned char byte;
...
byte b1, b2;
```

- Une autre utilisation fréquente consiste à rendre une programmation plus facile à adapter, ou à réserver des choix de représentation. Par exemple, un programme orienté calcul en flottants aura avantage à définir un type symbolique :

```
typedef float real;  /* Alias        */
...
real x, y;           /* Declarations */
...
```

Le jour où l'on souhaite passer l'application en version double précision, il suffira de changer une seule ligne de code,

```
typedef double real;
```

avant recompilation.

Dans le même ordre d'idée, lorsqu'on utilise des types non standard, comme **long long** ou **long double**, on aura avantage à les manipuler via un **typedef** pour permettre plus facilement des adaptations ultérieures.

- Enfin, on pourra plus facilement écrire des applications portables entre différentes machines. Par exemple, un programme devant manipuler des données entières sur 32 bits pourra définir :

```
typedef int int32;
```

pour des machines 32 bits, ou :

```
typedef long int32;
```

pour des machines 16 bits. A part cette adaptation, l'ensemble du code n'utilisera que le type **int32** pour ses variables et fonctions.

3.6 Initialisations

C autorise les déclarations de variables avec initialisation. Par exemple :

```
float x3 = 10.5;
int ii =1, jj = 2;
```

L'effet de ces initialisations dépend de la nature de la variable. Une variable glob-

ale, ou locale statique, est initialisée une fois pour toute en début d'exécution du programme. Mécanisme similaire au **data** de Fortran.

Une variable locale automatique est (re)initialisée à chaque création de son contexte, donc à chaque appel de la fonction qui la contient.

3.6.1 Types énumérés

Un outil très particulier, l'énumération, permet de construire et faire initialiser automatiquement, par le compilateur, une suite de constantes numériques symboliques.

Par exemple, un programme référence des couleurs par des codes numériques arbitraires. On écrira :

```
enum color { rouge, vert, jaune, noir, bleu, violet };
```

Le compilateur associe à chacun des identificateurs **rouge**, **vert**, etc., des valeurs numériques ordonnées : 0, 1, 2, 3, ...

On pourra ensuite déclarer des variables, de type "couleur énumérée", et utiliser ces constantes symboliques :

```
enum color toto;
toto = jaune;
...
if( toto < vert ) ...
```

Les variables déclarées ainsi sont des entiers, en général l'entier par défaut **int**. L'intérêt de ce mécanisme est d'une part de s'affranchir du travail fastidieux de définition de constantes arbitraires, d'autre part d'augmenter la lisibilité des programmes.

NB : on peut alléger les notations en utilisant l'alias de type (paragraphe 3.5 page 13) pour la déclaration des types énumérés :

```
typedef enum { rouge, vert, jaune, ... } color;
color toto;
```

Le symbole **color** est devenu un type à part entière, on peut déclarer des variables sans devoir spécifier **enum color**.

3.7 Fonctions du C-Ansi

La plupart des langages de programmation proposent deux ou trois sortes de modules de programme, **function** ou **procedure** en Pascal, **function**, **subroutine** ou **program** en Fortran. Le langage C ne propose que des fonctions. La notion de routine est vue comme une fonction particulière ne retournant rien ou dont on ignore la valeur retournée (cf paragraphe 3.9 page 16).

➤ Un *prototype* de fonction C est une déclaration indiquant le type, le nom et les types des arguments d'appel d'une fonction. Par exemple, une fonction calculant le minimum de deux réels se déclarera par :

```
float mini(float, float);       /* Prototype simple  */
float mini(float x, float y);   /* Arguments fictifs */
```

L'utilisation de noms d'arguments dans un prototype est optionnelle, et ne sert à rien

pour la compilation. Son seul intérêt est de documenter l'appel. Par exemple :

```
float fraction(float numerateur, float denominateur);
```

permet au programmeur n'ayant pas sous la main les spécifications d'une fonction de "deviner" l'ordre des arguments sur simple examen du prototype.

➤ La *définition* d'une fonction comporte l'interface d'appel, type et nom, les types et noms symboliques des arguments et le corps de fonction proprement dit :

```
float mini(float a, float b)
{
    if( a < b ) return a;
    else return b;
}
```

NB : On remarquera qu'en C la valeur retournée par une fonction se fait par une instruction spécifique **`return <valeur>`** et non par une affectation comme en Pascal, **`mini := <valeur>;`**, ou en Fortran, **`mini = <valeur>`**

Remarque : le prototype de fonction est un gage de sécurité dans les programmes C. L'exemple Fortran suivant illustre un piège classique dans lequel tout le monde est tombé au moins une fois[8] :

```
real x, y
real mini
...
x = mini(y, 5)
```

On a oublié d'informer le compilateur Fortran que la constante **`5`** devait être réelle, **`5.0`** ! Résultat fantaisiste assuré.

En **C-Ansi** :

```
float x, y;
float mini(float, float);
...
x = mini(y, 5);
```

le problème a disparu. Le compilateur, ayant vu le prototype, fera de lui-même la conversion de type correcte. De plus, si l'on appelle une fonction avec trop ou trop peu d'arguments par rapport au prototype, on aura un message d'erreur en compilation au lieu d'un crash en exécution.

Remarque : dans un même fichier source, une définition de fonction a valeur de prototype, une fois la définition traitée. Ainsi, dans l'exemple 2.2 page 5, la fonction **`linear3`** peut "appeler" la fonction **`linear1`** dans prototype puisque le compilateur a "vu" la définition de **`linear1`**.

Si l'on inversait l'ordre des définitions, **`linear3`** définie avant **`linear1`**, il faudrait ajouter un prototype :

```
float linear1(float);
```

avant la définition de **`linear3`**.

(8) Ne protestez pas, vous l'avez déjà fait, comme je l'ai déjà fait !

3.8 Fonctions du C-K&R

Dans le C de Kernighan et Ritchie, le prototype de fonction n'existait pas. Une déclaration se limitait au type :

```
float mini();
```

et une définition s'écrivait :

```
float mini(a, b)
float a, b;
{
    if( a < b ) return a;
    else return b;
}
```

Cette syntaxe est rappelée ici uniquement pour information, dans la mesure où l'on peut la rencontrer dans des sources C anciens.

Il faut donc la connaître de vue, mais ne jamais l'utiliser dans des nouveaux développements. Elle n'apporte strictement rien, et empêche le compilateur d'effectuer les contrôles d'appels. C est un langage synthétique et puissant, mais assez délicat à utiliser. Les probabilités de *bugs* en écriture sont loin d'être négligeables, et il n'y a aucune raison de les favoriser[9].

3.9 Pseudo fonctions

3.9.1 Type vide

On a déjà signalé que le langage C ne connait pas la notion de **`subroutine`** de Fortran, ou **`procedure`** de Pascal, ceci pour des raisons qui seront explicitées plus loin (cf. paragraphe 4.3 page 21).

Dans un programme, tout traitement passe donc par l'écriture d'une fonction. Lorsqu'on attend pas de résultat particulier en retour (traitement de type procédural), on peut se contenter d'ignorer la valeur retour de la fonction. Le **C-K&R**, assez laxiste en ce qui concerne la syntaxe, permettait de définir une fonction en omettant délibérément le type.

Le **C-Ansi**, beaucoup plus rigoureux, ne tolère plus ce genre d'approximation et a donc introduit un pseudo type **`void`** permettant de définir des fonctions sans type :

```
void routine(int arg)
{
    ...
}
```

La définition de la fonction ne devra pas comporter d'instruction **`return`**, et

(9) Sauf masochisme profond et relation affective quasi freudienne avec les debuggers symboliques.

l'invocation ne pourra être effectuée via une affectation :

```
float x;
...
routine(3);          /* Correct                */
...
x = routine(6);      /* Erreur en compilation */
```

NB : on remarquera que le mécanisme d'appel se fait par simple invocation, l'équivalent du **call** de Fortran n'existe pas en C.

Le spécificateur **void** sert également à décrire, au niveau du prototype ou de la définition, des fonctions sans arguments d'appel. Par exemple :

```
double random(void);     /* Prototype         */
double value;            /* Variable locale   */
...
value = random();        /* Appel             */
```

NB : l'appel d'une fonction sans arguments est possible, contrairement à Fortran, mais les parenthèses restent obligatoires.

3.9.2 Point d'entrée programme

De même que la notion de **subroutine** n'existe pas, en C, la notion de **program**, au sens Fortran, n'existe pas non plus.

La spécification du point d'entrée d'un programme C se fera en implantant une fonction particulière dont le nom est imposé :

```
int main(void)
{
    ...
    return 0;
}
```

Remarques :

- L'utilisation du nom réservé **main** a simplement pour but d'informer le compilateur que c'est cette fonction qui doit être activée lors du lancement du programme. Le compilateur prévoiera l'information ad-hoc, dans le fichier objet, pour l'éditeur de liens.
- Une et une seule fonction nommée **main** doit exister dans l'ensemble des modules de programme d'une même application.
- La fonction **main** est habituellement (architectures Unix) une fonction de type **int**, scalaire entier, qui doit retourner comme valeur le *compte rendu d'exécution* du programme.

  Sous Unix, un processus retourne 0 pour indiquer que tout s'est bien passé, ou un code erreur non nul en cas de problème. Il faut prendre l'habitude, en programmation C, de respecter cette convention qui permet en particulier de gérer correctement des exécutions depuis un fichier de commandes (*shell script*). En particulier, lorsqu'un programme est dans l'impossibilité de faire le travail prévu (erreur en ouverture d'un fichier de données ou autre problème de ce type), il faut quitter en retournant une valeur non nulle ! Voir en annexe G.1 page 72.
- La fonction **main** est, le plus souvent, une fonction sans arguments d'appels (cf.

l'écriture **`int main(void)`**). En toute rigueur on pourra utiliser le mécanisme d'appel pour récupérer des arguments de lancement (options de la ligne de commande). Cette possibilité est développée en annexe G.1 page 71.

3.10 Notes commentaires

[3.1] C'est vrai avec l'alphabet informatique standard Ascii et avec les alphabets européens ISO-8859-x.

L'internationalisation des applications informatiques a introduit un nouveau standard, l'Unicode, codant des caractères de toutes provenances (langages occidentaux, langages arabes et asiatiques) sur 16 ou 24 bits.

4 Opérateurs et expressions

Conçu initialement pour écrire le système d'exploitation Unix, C est un langage curieux. Sous un habillage respectable de langage évolué, il dissimule tout un ensemble de fonctionnalités proches de celles des assembleurs symboliques. (La notion de variable **`register`**, déjà signalée, en est un exemple.)

L'idée des concepteurs de C était de pouvoir disposer d'un langage de haut niveau autorisant des programmations presque aussi performantes qu'en assembleur natif.

En plus de tous les opérateurs classiques, on va donc trouver en C toute une famille d'opérateurs plus ou moins exotiques orientés machine.

4.1 Constantes numériques

4.1.1 Constantes entières

C permet de définir des constantes numériques entières sous différentes notations :

décimal Notation classique sous forme d'une suite de chiffres, le premier étant différent de zéro : **`12`**, **`45600`**, etc.

octal Suite de chiffres valides en base 8, précédée par un **`0`** initial : **`040`** (32 décimal), **`0377`** (255 décimal), etc.

hexadécimal Suite de chiffres valides en base 16, précédée par **`0X`** : **`0X10`** (16 décimal), **`0XFF`** (255 décimal), etc.

caractère Caractère alphabétique placé entre apostrophes, la signification numérique est le code Ascii : **`'A'`** (65 décimal), **`'0'`** (48 décimal).

NB : on a déjà signalé que le type **`char`** était bel et bien un type numérique entier, et nom un type caractère inexistant en C. De même, la notation caractère défini une constante numérique, avec la signification : "valeur numérique du code de ce caractère dans l'alphabet de la machine !" (en principe alphabet Ascii).

Ecrire **`i = 'A'`** ou écrire **`i = 65`** sont équivalents, cette notation n'est qu'une simplication.

NB : certains codes caractères sont d'usage relativement fréquent. En particulier, une application destinée à traiter du texte devra manipuler des codes de contrôle, fin de ligne, tabulation, saut de page, etc. Au lieu d'obliger le programmeur à connaître la table Ascii par coeur ou presque, C propose des notations symboliques spécifiques. Se reporter à l'annexe D.2 page 55.

4.1.2 Constantes longues

Par défaut, une constante numérique entière est de type **`int`**. On peut spécifier explicitement un type **`long`**, surtout utile sur des machines 16 bits, en ajoutant un **`L`**

à la définition :

```
int  ii = 35000;
long il = 35000L;
```

Cette notation, qui servait essentiellement à expliciter des appels de fonctions avec arguments longs, est de moindre utilité en **C-Ansi** où le compilateur effectue toutes conversions utiles à partir des prototypes de fonctions.

4.1.3 Constantes réelles

C utilise les notations traditionnelles, virgule flottante ou exponentielle : **`0.516`**, **`12.4E-3`**, etc.

Par défaut, une constante réelle (comportant un point décimal) est de type **`double`**. Il n'y a pas, en **C-Ansi**, de distinction entre **`float`** et **`double`**[10].

4.2 Expressions arithmétiques

4.2.1 Opérateurs arithmétiques

C implante les cinq opérateurs classiques de l'arithmétique sous les notations habituelles :

+	addition
-	soustraction (opérateur binaire)
-	ou inversion de signe (opérateur unaire)
*	multiplication
/	division
%	modulo

C propose également des opérateurs spéciaux d'incrémentation ou décrémentation (voir note [4.1] page 27) :

++	incrémentation
--	décrémentation

A la différence des autres, ces opérateurs ne sont applicables qu'à des variables et non à des expressions. Ecrire : **`n++`** est licite et incrémente la valeur de la variable **`n`** de 1, alors qu'écrire **`5++`** n'a aucun sens.

Lorsqu'une variable fait l'objet d'une incrémentation et, en même temps, apparait dans une expression, ces opérateurs s'utilisent en préfixe ou postfixe selon le résultat à obtenir :

```
int a, b;
b = 5;
a = ++b + 1;
b = 5;
a = b++ + 1;
```

Dans les deux exemples d'expressions ci-dessus, la variable **`b`** a pris la valeur **`6`** par

(10) Ce n'est plus vrai en C++, et certains compilateurs C compatibles C++ acceptent une notation explicite **`float`** par ajout d'un **`F`** à la constante. Par exemple : **`10.56F`**

incrémentation. Dans la première expression, **b** est *préincrémenté*, donc avant utilisation, et l'expression affectée à **a** vaut **7**. Dans la seconde, **b** est *postincrémenté*, donc après utilisation, et **a** vaudra **6** !

4.2.2 Opérateurs binaires

Toujours dans le même esprit d'efficacité (voir note [4.2] page 27 en fin de chapitre), C propose également des opérateurs destinés à manipuler des données au niveau du bit[11] :

~	complément à 1 de tous les bits d'une donnée : **~7** vaut **0XF8** sur 8 bits, **0XFFF8** sur 16 bits, etc.
&	**AND** bits à bits : **5 & 4** vaut **4**.
|	**OR** bits à bits : **5 | 3** vaut **7**.
^	**XOR** (ou exclusif) bits à bits : **5 ^ 3** vaut **4**.
<<	décalage de bits à gauche : **5 << 2** vaut **20**.
>>	décalage de bits à droite : **8 >> 1** vaut **4**.

4.3 Expressions généralisées

4.3.1 Affectations

L'affectation à une variable du résultat d'une expression existe dans tous les langages, avec de légères variations de syntaxe, **n := 15** en Pascal, **n = 15** en Fortran.

L'affectation en C, symbolisée par **=**, n'est pas une instruction comme dans les autres langages mais un opérateur arithmétique ! Ainsi :

```
n = 15;
```

affecte à **n** la valeur **15** et retourne comme résultat son opérande de droite, ici **15**.

Ce qui ne semble être qu'un détail de folklore est très riche de conséquences :

- Une expression arithmétique est une composition d'opérateurs et d'opérandes, en nombre a priori quelconque, avec des règles d'évaluation (priorités et associativités).

  L'affectation étant un opérateur, une expression telle que :

  ```
  a = b = c = 1
  ```

  est tout à fait licite, au même titre que :

  ```
  a + b + c
  ```

  On peut donc effectuer des "affectations en chaîne", et même construire des expressions telles que :

  ```
  a = (b = 10) + 2;
  ```

  qui affecte à **b** la valeur **10** et à **a** la valeur **12**. Les parenthèses ici ont un rôle purement sémantique, l'opérateur **=** étant moins prioritaire que l'opérateur **+**.

(11) Binaire s'entend ici au sens bit, et non au sens opérateur à deux opérandes !

- Lorsque l'on effectue une affectation simple :

  ```
  n = 25;
  ```

  on écrit en réalité une expression arithmétique qui vaut **25**. C autorise ce genre de chose, et des écritures telles que :

  ```
  n + 25;
  ```

  ou, encore plus minimaliste :

  ```
  25;
  ```

  sont tout aussi licites, même si elles n'ont aucun intérêt !

  En poussant la logique encore plus loin, on peut dire qu'un appel de fonction est une expression arithmétique atomique, à valeur la valeur de retour de la fonction. Par exemple :

  ```
  sqrt(50.0);
  ```

  est une écriture C tout à fait valide.

En programmation C, tout passe donc par des expressions généralisées, suite d'opérateurs et d'opérandes, terminée par **;** La valeur résultante de l'évaluation de l'expression est tout simplement ignorée.

C'est ce qui explique que ce langage n'a pas besoin, contrairement aux autres, de faire une distinction entre des modules de code de type *fonction* et des modules de type *subroutine*, puisqu'un appel à routine (le **call** de Fortran) n'est qu'une invocation de fonction dont on ignorera la valeur de retour. Le pseudo type **void** n'a, en réalité, qu'un intérêt syntaxique.

4.3.2 Affectations composées

Dans le même ordre d'idées que les opérateurs spécialisés d'incrémentation et de décrémentation, C propose des affectations composées, l'opérande de gauche devant être une variable.

Au lieu d'écrire :

```
n = n + 4;
```

on pourra écrire :

```
n += 4;
```

Tous les opérateurs arithmétiques et binaires déjà présentés existent en version "affectation composée" : **+=**, **-=**, ***=**, **/=**, **%=**, **~=**, **&=**, **|=**, **^=**, **<<=** et **>>=**.

4.3.3 Jeu

(Pour détendre une atmosphère que l'on devine quelque peu crispée !)

Sans tricher (sans écrire ni faire tourner un programme C donnant la solution), trouver combien valent les variables **a**, **b** et **c** après l'expression généralisée suivante :

```
a += ' ' + b++ + (a = b = c = 2);
```

4.4 Expressions booléennes

Il n'existe pas, en C, de type spécifique booléen analogue au **boolean** de Pascal ou au **logical** de Fortran. La notion de "vrai ou faux" est implantée par le biais d'expressions arithmétiques respectivement "non nulles" ou "nulles".

Ceci peut conduire parfois à des écritures déroutantes pour le néophyte. Ainsi l'écriture suivante :

```
if( x - 3 ) ...
```

doit se lire "si l'expression **x - 3** est non nulle, i. e. vraie", ce qui revient à dire "si **x** est différent de **3**".

De même, une écriture telle que :

```
while( x ) ...
```

est la traduction (un peu surprenante) de "tant que **x** est non nul".

NB : ce genre d'écriture est à utiliser avec des variables ou expressions entières, la notion de réel nul étant informatiquement floue !

4.4.1 Opérateurs booléens

À part ce détail d'implantation, C dispose des classiques comparateurs et opérateurs booléens permettant de construire tout type d'expression logique :

<	comparateur "inférieur".
<=	comparateur "inférieur ou égal".
>	comparateur "supérieur".
>=	comparateur "supérieur ou égal".
==	comparateur "égal".
!=	comparateur "différent".
&&	connecteur booléen "ET".
||	connecteur booléen "OU".
!	connecteur booléen "NON"

Remarque : la richesse de C, en mécanismes opératoires, fait qu'il existe très souvent plusieurs écritures possibles pour traduire la même chose. Ainsi :

```
if( n == 0 ) ...
```

est l'écriture conventionnelle pour dire "si **n** est égal à 0", mais on trouvera souvent, dans des sources C, des écritures comme :

```
if( ! n ) ...
```

qui est la traduction de "si l'expression **n** est fausse", i.e. nulle.

4.4.2 Pièges syntaxiques

On a montré, au paragraphe 4.3 page 21, que la notion d'expression généralisée est très riche de possibilités. Elle est également très riche en tant que source d'erreurs !

On peut dire que pratiquement toute écriture C a une sémantique possible. De ce fait, on commet souvent des erreurs de programmation que le compilateur ne détectera pas parce que l'écriture correspond à quelque chose de plausible.

Une erreur très fréquente[12] est la confusion entre la comparaison **==** et l'affectation **=**. On a voulu écrire :

```
if( n == 0 ) ...
```

et on a écrit par erreur :

```
if( n = 0 ) ...
```

La seconde écriture "passe" très bien en compilation car elle est tout à fait correcte : **n = 0** est une expression qui affecte à **n** la valeur **0** et retourne un résultat, ici nul, valide dans un contexte booléen.

En exécution, d'une part on a détruit la valeur de la variable **n**, que l'on voulait simplement tester, en lui affectant une valeur nulle, et d'autre part on a construit une expression à valeur nulle, donc évaluée à "faux" d'un point de vue booléen. Le code associé au **if** ne sera jamais exécuté !

Cette erreur est tellement classique (et pas facile à identifier après coup au travers des bizarreries de fonctionnement) que certains compilateurs compatissants la surveillent. Ainsi, le GNU-C (voir 1.1 page 1) affiche un message *warning* et demande que les affectations effectuées dans des contextes booléens soient explicitées par des parenthèses. Il faudrait écrire :

```
if( (n = 0) ) ...
```

pour dire au compilateur : "Oui ! Je sais ce que je fais !"

Ce type d'erreur se produit avec tous les opérateurs qui ont une symbolique voisine et des effets différents. Ne pas confondre **&&**, **AND** booléen, qui réalise le **ET** logique de deux opérandes booléens, avec **&**, **AND** binaire, qui effectue une opération bits à bits.

4.5 Évaluations

4.5.1 Priorités

Les expressions, en C comme dans les autres langages, sont évaluées à partir de règles de priorité des opérateurs, opérateurs multiplicatifs plus prioritaires que les opérateurs additifs, comparateurs plus prioritaires que les opérateurs booléens, etc., et à partir de régles d'associativité, gauche à droite ou droite à gauche.

On modifie les règles de priorité en incorporant des parenthèses de ponctuation dans

(12) Par très fréquente, il faut lire : erreur que quiconque ayant déjà programmé en C a commise, et que quiconque envisageant de programmer en C va commettre ! C'est prouvé !

les expressions. Ainsi, le produit de deux sommes doit s'écrire :

```
(a + b) * (c + d)
```

au lieu de :

```
a + b * c + d
```

la multiplication étant plus prioritaire que l'addition.

Un opérateur comme l'affectation a une priorité plus basse que les opérations arithmétiques, c'est pourquoi dans un exemple déjà vu on devait écrire :

```
a = (b = 10) + 2;
```

et non :

```
a = b = 10 + 2;
```

De plus, cet opérateur est associatif de droite à gauche, ce qui est logique compte tenu de son utilisation potentielle pour faire des affectations à la chaîne. Les opérateurs arithmétiques, eux, suivent en général la convention classique d'associativité de gauche à droite.

L'annexe C page 54, synthétise toutes les priorités et associativités des opérateurs du C. S'y reporter en cas de doute.

4.5.2 Conversions de type

Dans le traitement d'expressions hybrides, avec opérandes de nature différentes, C applique des mécanismes de conversion implicite de type.

Les règles sont classiques et visent à conserver la précision maximale des calculs : conversion en réel si un au moins des opérandes est réel, conversion sur le type d'entier le plus grand en cas d'expression mettant en oeuvre des entiers de taille différente, etc.

On peut forcer des conversions explicites par utilisation de constantes numériques typées :

```
int n = 10;
float x, y;
...
x = n / 3;      /* Division entiere -> 3     */
y = n / 3.0;    /* Division reelle -> 3.3333 */
```

On peut également utiliser un *cast*. Le cast est tout simplement une spécification de type placée entre parenthèses devant l'opérande visé. L'exemple ci-dessus peut s'écrire :

```
y = (float)n / 3;
```

n, entier, est converti (on dit : "est casté") en réel, donc la division se fera entre réels.

De même, on peut faire des opération entières explicites avec des opérandes réels :

```
z = (int)x / (int)y;
```

NB : le **cast** s'applique à un opérande bien précis. Pour *caster* des expressions, on

devra utiliser des parenthèses.

```
y = (float)n / 3;
```

n'est pas du tout la même chose que :

```
y = (float)(n / 3);
```

NB : le **cast** est parfois nécessaire dans des expressions logiques, en particulier des comparaisons de données de nature différente : entiers avec réels, entiers signés avec entiers non signés, etc. :

```
int ii;
unsigned jj;
...
if( ii < (int)jj ) ...       /* Comparaison signee */
if( (unsigned)ii < jj ) ... /* ou non signee      */
```

4.6 Opérateurs spéciaux

Enfin, C dispose de quelques opérateurs un peu exotiques qui n'ont rien de fondamental mais peuvent rendre service dans un certain nombre de cas.

4.6.1 Taille des données

L'opérateur **sizeof** permet de déterminer la taille informatique, en octets, d'une donnée. Il est utilisable avec un spécificateur de type : **sizeof(double)** vaut 8 (octets), **sizeof(short)** vaut 2, ou avec un identificateur de variable, **sizeof(x2)**.

Cet opérateur présente un intérêt limité pour des types ou variables scalaires, hormis quelques cas où l'on souhaiterait déterminer la taille des mots machine en exécution :

```
if( sizeof(int) == 2 ) ... Machine 16 bits
```

Il est très utile, par contre, pour des structures de données et tableaux, en particulier lorsqu'on manipule des zones mémoire ou lorsque l'on effectue des entrées/sorties fichiers binaires. Ces utilisations seront détaillées aux chapitres correspondants.

4.6.2 Séquenceur

Cet opérateur , (oui ! virgule), un peu artificiel, évalue ses deux opérandes et retourne la valeur de son opérande de droite. Ainsi :

```
a , b
```

vaut la valeur de **b**.

Son intérêt est de permettre certaines écritures qui seraient syntaxiquement impossibles, en particulier lorsque l'on a besoin d'effectuer plusieurs choses dans un contexte demandant une expression unique. Par exemple, une répétition :

```
while( i++, j--, j != 0 ) ...
```

L'écriture **i++, j--, j != 0** est une expression généralisée qui incrémente **i**, décrémente **j**, compare **j** à **0** et retourne une valeur arithmétique nulle ou non nulle,

donc licite comme élément booléen d'une conditionnelle.

Le lecteur attentif avait, bien sûr, tout de suite remarqué qu'une telle expression pouvait s'écrire de manière plus expéditive par :

```
while( i++, --j ) ...
```

4.6.3 Jeu

(Pour détendre etc.)

Trouver pourquoi l'écriture ci-dessus ne devait pas être écrite :

```
while( i++, j-- ) ...
```

4.6.4 Sélecteur

Cet opérateur **?** **:** permet d'implanter une évaluation conditionnelle dans une expression généralisée. La syntaxe est :

```
a ? b : c
```

le premier opérande, **a**, est évalué et interprété au sens booléen nul/non nul. S'il est vrai (non nul), l'expression vaut le second opérande **b**, sinon l'expression vaut le troisième opérande **c**.

Pratiquement, il s'agit donc d'un mécanisme de type **if ... else** mais utilisable dans un contexte où une structure de contrôle ne serait pas acceptée.

Par exemple, appel d'une fonction avec en argument la valeur absolue d'une variable :

```
y = truc( x < 0 ? -x : x );
```

notation moins lourde qu'une écriture algorithmique plus traditionnelle :

```
if( x < 0 ) y = truc(-x);
else y = truc(x);
```

En particulier, cet opérateur est très utile en macro programmation. Voir en annexe A.2 page 45.

4.7 Notes commentaires

[4.1] Ces opérateurs ne sont pas de simples gadgets. On peut estimer qu'écrire **n++** ou écrire plus classiquement **n = n + 1** ne change pas grand chose. Ces opérateurs font partie de la recherche de performance du langage, déjà évoquée, et un compilateur pourra les implanter de manière très efficace en s'appuyant sur des instructions machines spécifiques.

A fortiori lorsque ces opérateurs sont appliqués à des variables **register**.

[4.2] Il est un peu illusoire de penser qu'un bon compilateur optimiseur peut trouver de lui-même tous les trucs et astuces nécessaires à une traduction machine efficace.

D'une part, les heuristiques d'optimisation portent essentiellement sur des mécanismes algorithmiques : ouvertures de boucles, identification d'expressions constantes, etc. que sur la traduction en instruction machine. D'autre part un compilateur ne

peut jamais tout deviner, et le meilleur optimiseur qui soit restera encore longtemps l'être humain.

C'est pourquoi C propose des outils de programmation intelligente, plutôt que de s'appuyer sur les hypothétiques performances de tel ou tel compilateur.

5 Contrôles d'exécution

C dispose des structures de contrôle d'exécution classiques des langages pascaliens : alternatives, répétitions, sélections, etc.

5.1 Généralités

Comme en C la présentation du code est libre, passages à la ligne où l'on veut et quand on veut, les structures de contrôle doivent respecter quelques règles syntaxiques précises :

- Les structures de contrôles sont construites à partir de mots-clés du langage : **`if`**, **`else`**, **`do`**, **`while`**, etc.
- Ces structures comportent des expressions booléennes, au sens de C (expression arithmétiques nulles ou non nulles), qui doivent toujours figurer entre parenthèses !

  ```
  if( x < 0.0 ) ...
  ```

  ou

  ```
  while( i > j && k != 0 ) ...
  ```

  etc.
- Les traitements associés aux conditions sont ou bien une expression généralisée C, avec son terminateur `;` obligatoire, ou bien un contexte d'exécution placé entre accolades `{ ... }` et qui ne doit PAS être suivi d'un `;`

  Un exemple, déjà évoqué, peut s'écrire :

  ```
  if( x < 0 ) y = truc(-x);
  else y = truc(x);
  ```

  ou encore :

  ```
  if( x < 0 ) { y = truc(-x); }
  else { y = truc(x); }
  ```

Le choix de l'une ou l'autre forme, expression généralisée ou contexte, dépend des traitements à effectuer. L'utilisation d'un contexte autorise tout traitement, plus ou moins long, et comportant autant d'expressions ou autres structures de contrôle que nécessaire.

De plus, et c'est une particularité intéressante de C, l'utilisation d'un contexte permet

la définition de variables locales à ce contexte. Par exemple :

```
if( x < 0 ) {
   float xx, yy;
   yy = truc(x);
   xx = yy + ... ;
   ...
}
else .... ;
```

Les variables **xx** et **yy** appartiennent à un contexte local qui est créé uniquement si la condition évaluée par le **if** est vraie. Ces variables sont automatiques, et détruites lorsque le contexte se termine. Si, en exécution, la condition est fausse, ces variables n'auront même pas existé !

C permet donc de faire une gestion extrêmement fine des données en permettant de reporter certaines définitions uniquement là où elles sont nécessaires. De plus, on améliore la lisibilité et la maintenance des programmes en ne définissant en tête des fonctions que les variables locales nécessaires à l'ensemble du code de la fonction, et en reportant aux niveaux des contextes internes toutes les définitions occasionnelles.

Les règles de visibilité des identificateurs s'appliquent également aux contextes internes :

```
float truc(float xx)
{
   float yy;         /* Variable locale fonction */
   ...
   if( xx < 0 ) {
      float yy, zz; /* Variables locales contexte */
      ...
   }
```

La définition **yy** dans le contexte du **if** masque la variable **yy** déclarée en tête de fonction.

5.2 Alternative

- L'alternative existe, en C, sous les deux formes habituelles, simple :

```
if( <condition> ) <traitement>
```

- ou complète :

```
if( <condition> ) <traitement1>
else <traitement2>
```

L'expression **<condition>** est arithmétique à valeur booléenne, nulle ou non nulle, les traitements sont des expressions généralisées (avec terminateur **;**) ou des contextes entre accolades (sans **;** après l'accolade fermante).

On peut imbriquer des alternatives, dans ce cas les **else** sont toujours associés au **if** le "plus proche" :

```
if( <cond1> ) if( <cond2> ) ....;
else ....;
```

Le **else** se rapporte à **<cond2>** et non à **<cond1>**. En cas de doute, il est conseillé d'utiliser des contextes qui lèvent les ambiguïtés :

```
if( <cond1> ) {
   if( <cond2> ) ....;
   else ....;
}
```

5.3 Répétitions

➤ La répétition de base, en C, est le "Tant que" pascalien :

```
while( <condition> ) <expression>;
while( <condition> ) { ... }
```

où **<condition>** est une expression arithmétique à valeur booléenne et le traitement associé une expression généralisée ou un contexte.

Si l'expression est fausse dès la première évaluation, le traitement n'est pas effectué.

➤ Il existe la variante :

```
do <expression>;
while( <condition> );
do { ... }
while( <condition> );
```

qui permet d'effectuer le traitement au moins une fois, la condition étant évaluée après coup.

NB : on remarquera la présence du **;** final, après la parenthèse fermant la condition.

Attention : les habitué(e)s du langage Pascal se méfieront :
le '**do .. while**' de C est analogue au '**repeat .. until**' de Pascal, sous réserve d'inverser la valeur de la condition !

➤ En parallèle du mécanisme d'évaluation, il est possible de quitter prématurément une répétition '**while**' ou '**do .. while**' à l'aide de l'instruction **break** :

```
while( <condition> ) {
   ...
   if( ... ) break;
   ...
}
```

Même si ce mécanisme n'est pas "algorithmically correct", c'est parfois utile pour certains traitements. Voir note [5.1] page 34 en fin de chapitre.

➤ Enfin, une dernière variante permet d'implanter les boucles indicées traditionnelles :

```
for( <expr1> ; <expr2> ; <expr3> ) <expression>;
for( <expr1> ; <expr2> ; <expr3> ) { ... }
```

Cette structure est très générale, dans sa construction, et autorise tous les cas de figure possibles selon le schéma suivant :

1. L'expression **<expr1>** est évaluée une seule fois, en début de boucle. C'est l'initialisation.

2. L'expression **<expr2>** est évaluée dans un sens booléen et sert de condition de répétition.
3. Le traitement, expression généralisée ou contexte est effectué.
4. L'expression **<expr3>** est évaluée. C'est l'incrémentation. Ensuite, on reprend en 2.

NB : l'ordre de ces opérations est important. En particulier, si la condition de répétition est fausse d'emblée, aucun traitement n'est effectué.

Par exemple, une classique boucle indicée, de 1 à 20, peut s'écrire :

```
for( i = 1 ; i <= 20 ; i++ ) ...
```

Les parties "initialisation" et "incrémentation" étant des expressions généralisées, a priori quelconques, on peut écrire des boucles plus élaborées :

```
for( i = 0, j = 2 ; i <= 10 * j ; i += 3, j++ ) ...
```

Enfin, les trois expressions de contrôle de boucle sont optionnelles, ce qui fait que le **for** permet d'écrire à peu près n'importe quoi, une répétition de type **while**, en omettant les première et dernière expressions :

```
for( ; <condition> ; ) ...
```

ou même une boucle infinie, l'absence de la deuxième expression ayant pour valeur "vrai" :

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

Dans ce cas, il ne faudra pas oublier de prévoir un moyen de quitter la boucle, par exemple via un **break**.

5.4 Sélection

➤ C dispose d'une sélection, implantée de la manière suivante :

```
switch( <expression> ) {
case <valeur1> :
    ... traitement 1
case <valeur2> :
    ... traitement 2
default :
    ... traitement par defaut
}
```

L'expression objet de la sélection, **<expression>**, est évaluée est comparée aux valeurs des clauses **case**, **<valeur1>**, **<valeur2>**, etc., qui doivent obligatoirement être des constantes numériques ou des types énumérés (cf. paragraphe 3.6 page 14).

Lorsqu'une comparaison est vraie, le traitement associé est exécuté. Si aucune comparaison n'est vérifiée, le traitement associé à la clause **default** est exécuté. Cette clause par défaut est optionnelle. Le délimiteur **:** est obligatoire après les clauses !

Attention : contrairement à d'autres langages (Pascal), en C une clause de sélection défini un point d'entrée dans les traitements qui sont ensuite exécutés en séquence et pas limités à la clause.

Ainsi, dans l'exemple ci-dessus, si **<expression>** vaut **<valeur1>**, on exécutera le traitement 1 mais aussi le traitement 2 et tous les suivants jusqu'au traitement par défaut inclus ! Si l'on veut définir des traitements associés à une valeur de clause et une seule, on quittera la sélection prématurément :

```
switch( <expression> ) {
case <valeur1> :
    ... traitement 1
    break;
case <valeur2> :
    ... traitement 2
    break;
...
```

Enfin, on peut également regrouper plusieurs valeurs de sélection pour le même traitement :

```
case <valeur1> :
case <valeur2> :
case <valeur3> :
    ... traitement 1
    break;
case <valeur4> :
case <valeur5> :
    ... traitement 2
    break;
...
```

5.5 Branchement

➤ Enfin, C comporte une instruction de branchement utilisant des étiquettes symboliques alphanumériques :

```
goto ailleurs;
...
ailleurs$_:
...
```

NB : la définition du point de branchement doit comporter le délimiteur **~:** après le nom.

Ce mécanisme de contrôle d'exécution, souvent décriés par les tenants d'une programmation pascalienne stricte est, en pratique, de peu d'utilité. C est suffisament riche en structures de contrôle pour qu'on ne soit jamais obligé de programmer à grands renforts de **goto**.

Il faut savoir qu'il existe car il peut être à l'origine d'une erreur lors de l'utilisation

d'une sélection :

```
switch( <expression> ) {
case <valeur1> :
     ...
     ...
defaut :
     ...
}
```

Dans cet exemple, on a écrit **defaut** au lieu de **default** ! Cette erreur passe inaperçue à la compilation puisque, syntaxiquement parlant, la présence du caractère **:** fait que **defaut** peut être une étiquette de branchement. Certains compilateurs scrupuleux afficheront un message signalant que **defaut** n'est pas utilisée, i. e. aucune instruction '**goto defaut;**' n'existe dans le code source.

La plupart des compilateurs ne diront rien du tout, et on pourra mettre un moment à chercher pourquoi, en exécution, le traitement par défaut prévu n'est jamais exécuté.

Ce *bug* potentiel ne concerne, bien sûr, que les programmeurs francophones.

5.6 Notes commentaires

[5.1] On a déjà signalé le fait que C est un langage hybride, langage pascalien évolué et, en même temps, comportant des fonctionnalités proches de celles des macro assembleurs.

Le souci constant d'efficacité, chez les concepteurs de ce langage, a conduit à faire cohabiter des structures algorithmiques pascaliennes strictes avec des fonctionnalités plus expéditives. Le **break** en est un exemple, ce n'est rien d'autre qu'un "vulgaire" saut, "go to fin de boucle" !

De fait, C a souvent été critiqué par les tenants d'une école pascalienne pure et dure comme étant un "langage de bidouilleurs" ! Sans rentrer dans ce débat, de peu d'intérêt, on pourra dire qu'un langage n'est qu'un support, qu'on n'est jamais obligé d'en utiliser toutes les possibilités, et qu'un bon programme est un programme qui fonctionne bien, qui est clair, lisible et convenablement documenté, et qui est facile à maintenir et adapter.

6 Pointeurs

Dans tout programme, quelque soit le langage de programmation utilisé, les variables sont rangées en mémoire à des *adresses* plus ou moins arbitraires.

Lorsque l'on utilise des langages évolués, Fortran, Pascal, C, les variables sont référencées, dans les fichiers source, par des noms symboliques. Le compilateur se chargera de construire des tables de correspondance symbole–adresse qui seront mises à jour au cours de la phase d'édition de liens.

Ce mécanisme est, dans la plupart des langages, transparent au programmeur. En C, on peut manipuler explicitement des adresses de variables, et même créer des variables spécialisées, destinées à recevoir et manipuler des adresses, nommées *pointeurs*.

6.1 Adresses

Après avoir déclaré et typé une variable :

```
int toto;
```

l'utilisation du symbole **toto**, dans des expressions, signifie par convention "valeur de la variable **toto**".

- La notation symbolique :

```
&toto
```

signifie "adresse de la variable **toto**". C'est une quantité numérique entière, adresse mémoire, dont la taille est spécifique à la machine[13].

- Un pointeur de variable se déclare de la manière suivante :

```
int   *pi;
float *pf;
```

Le caractère *****, placé devant le nom, indique qu'il s'agit d'un pointeur. Le type spécifie la nature de la donnée pointée. Un pointeur est, en effet, typé. Dans l'exemple, **pi** est un pointeur d'**int**, c'est à dire une variable destinée à manipuler des adresses de variables de type **int**, et **pf** est un pointeur de **float**, destiné à manipuler des adresses de variables de type **float**.

- La valeur d'un pointeur, après déclaration, est a priori indéterminée (au même titre que la valeur d'un scalaire). On lui affectera une valeur, typiquement, en prenant

(13) Et pas obligatoirement de la même taille que l'entier par défaut. Ainsi, sur les machines DEC-Alpha, l'entier par défaut **int** fait 32 bits alors que les adresses font 64 bits.

l'adresse d'une autre variable de nature convenable :

```
int ii, jj; /* Declarations de scalaires    */
float x;
int   *pi;  /* Declarations de pointeurs    */
float *pf;
pi = &ii;   /* Affectations de pointeurs    */
pf = &x;
```

➤ Une fois initialisé sur une adresse valide, un pointeur permet de manipuler la donnée pointée à l'aide de l'opérateur ***** appelé opérateur d'indirection :

```
*pf = 0.5;       /* Affecte a x la valeur 0.5        */
jj = 2 + *pi;    /* Affecte a jj la valeur de ii + 2 */
```

Les notations ***pi** ou ***pf** se lisent "contenu de pi ou pf".

Ce mécanisme d'indirection explique pourquoi un pointeur doit être typé : le compilateur a besoin de savoir comment interpréter le "contenu de", entier court, long, réel simple ou double précision.

6.1.1 Limitations

- On ne peut pas utiliser l'opérateur adresse **&** avec des constantes numériques. L'écriture **&10** n'a aucun sens.
- On ne peut pas utiliser l'opérateur adresse **&** avec une variable déclarée en classe **register**. Une variable registre est placée dans un des registres internes du CPU, lesquels ne sont pas numériquement adressables.
- On ne doit manipuler par pointeur que des données du type ad-hoc, homogène avec le typage du pointeur[14]. L'inverse serait en tout état de cause dangereux et source de problème. On peut toutefois contourner cette restriction en utilisant un **cast** (cf. paragraphe 4.5 page 25) explicite sur le pointeur.

  Par exemple :

  ```
  int ii;
  int *pi;
  short si;
  pi = &ii;
  si = *((short*)pi);
  ```

  La notation **(short*)pi** est un **cast** qui transforme pi en pointeur de **short**. L'indirection ***((short*)pi)** accède donc à l'adresse mémoire "valeur de pi" interprétée comme une adresse de variable **short**.

  Attention : lorsque l'on joue avec ce genre de chose, il faut savoir ce que l'on fait ! En particulier, l'ordre de rangement en mémoire des octets d'un numérique dépend de la machine : du plus bas au plus fort sur les processeurs *little endian* (DEC Alpha, Intel Pentium) ou du plus fort au plus bas sur les processeurs *big endian* (Sun Sparc, Apple PPC). Le même code symbolique C peut donc donner des résultats complètement différents selon la plateforme.

(14) Et le compilateur y veille !

6.2 Passages d'arguments

Lorsqu'on appelle une routine ou fonction, dans un programme, les arguments d'appel sont transmis selon des mécanismes propres au langage de programmation utilisé.

Le langage Fortran utilise le mécanisme dit "par références" qui consiste à transmettre à la routine appelée les adresses mémoire des arguments. Par exemple une routine destinée à permuter les valeurs de deux entiers s'écrira :

```
subroutine swap(a, b)
integer a, b
integer temp
temp = a
a = b
b = temp
end
```

et s'utilisera par :

```
integer ii, jj
...
call swap(ii, jj)
```

Ce mécanisme ne fonctionne pas du tout en C qui utilise le passage dit "par valeurs" consistant à transmettre à la routine appelée une copie des arguments d'appel. Cette copie sert, de plus, à initialiser les variables arguments qui sont des variables locales automatiques, associées au contexte. (Revoir le paragraphe 2.2 page 6 à ce sujet.)

Ce mécanisme apporte une réelle sécurité, en empêchant la modification de données appartenant au contexte appelant suite à des erreurs de programmation dans une routine. Par contre, il est blocant dans les quelques cas où l'on voudrait effectivement modifier le contexte appelant.

La solution consiste à utiliser des adresses de données explicites. L'exemple ci-dessus s'écrira, en C :

```
void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

et s'utilisera par :

```
int ii, jj;
...
swap(&ii, &jj);
```

La fonction **swap** est construite pour recevoir des arguments qui sont des adresses (utilisation de l'opérateur d'indirection dans la permutation). Elle est appelée en passant explicitement les adresses des variables à permuter.

6.3 Tableaux

6.3.1 Les tableaux existent, en C

C permet de déclarer des tableaux de variables, avec une syntaxe assez classique :

```
float tabx[50];      /* Declaration, dimension 50    */
int i;
tabx[20] = 0.5;      /* Affectations */
for(i = 21; i < 30; i++) tabx[i] = tabx[i - 1];
...
```

Attention : en C, les tableaux sont TOUJOURS indicés à partir de 0 ! Le tableau ci-dessus est utilisable de **`tabx[0]`** à **`tabx[49]`**. Les habitué(e)s de Fortran vont devoir faire preuve de vigilance.

Les matrices et autres variables indicées à plusieurs dimensions n'existent pas en tant que telles et sont implantées par des tableaux de tableaux de tableaux ... comme le suggère la syntaxe suivante :

```
float mat[10][20][30];  /* (Et non : mat[10,20,30] !) */
...
mat[1][2][3] = 0.0;
```

NB : ce mécanisme simple élimine toute restriction relative au nombre maximal de dimensions d'une matrice. Il a une conséquence sur le rangement en mémoire des éléments, voir la note [6.1] page 43 en fin de chapitre.

6.3.2 Les tableaux n'existent pas, en C

Contrairement à une croyance répandue, et à ce qui a été dit ci-dessus, les tableaux de variables n'existent pas en C. Les notations ci-dessus ne sont qu'une simplification syntaxique, l'implantation effective passe par des pointeurs initialisés sur une zone mémoire de taille convenable.

➤ Le nom symbolique du tableau est homogène à une variable de type pointeur, préinitialisée sur l'adresse du premier élément :

```
float tabx[50];      /* "Tableau" de float   */
float *pf;           /* Pointeur de float    */
pf = tabx;
*pf = 0.5;
pf[10] = 10.5;
*(pf + 20) = 12.5;
```

L'affectation '**`pf = tabx`**' est tout à fait licite puisque **`pf`** et **`tabx`** sont tous deux des pointeurs de **`float`**.

La notation **`*pf`**, "contenu de pf", référence le premier élément, soit **`tabx[0]`**.

La syntaxe **`[..]`** est valide, même avec des pointeurs simples (non déclarés sous la forme tableau). La notation **`pf[10]`** est équivalente à **`tabx[10]`**. De même, on aurait pu écrire **`pf[0]`** au lieu de **`*pf`**.

Un *déplacement* est autorisé dans les accès par pointeurs. La notation **`*(pf + 20)`**

signifie "contenu de l'adresse mémoire obtenue à partir de **pf** augmentée de 20". Cette notation est strictement équivalente à **pf[20]**.

Important : les notations arithmétiques utilisées avec des pointeurs sont basées sur la taille des éléments pointés. Écrire **pf + 20** déplace la valeur de **pf** de 20 fois la taille d'un **float**, soit 80 octets mémoire.

Avec un pointeur d'entiers 8 bits :

```
char *pc;
```

écrire **pc + 20** déplacerait la valeur de **pc** de 20 fois un octet. C'est une autre des raisons pour lesquelles le typage des pointeurs est obligatoire.

Remarque : la notation **[..]** dissimule, en réalité, ce mécanisme d'accès par adresse base et déplacement. C'est ce qui explique pourquoi le premier élément d'un tableau C est indicé par 0. Il est rangé à l'adresse base du tableau décalée de ... rien du tout !

➤ Enfin, lorsque l'on doit "passer" un tableau en argument d'appel à fonction, on écrira l'appel :

```
machin(tabx);
```

où **tabx** est, en fait, un pointeur, et on implantera la fonction par :

```
void machin(float *tab)
{
    ...
}
```

6.3.3 Tableaux initialisés

De même qu'il est possible d'initialiser une variable C lors de la déclaration (cf. paragraphe 3.6 page 13), on peut déclarer et initialiser un tableau de la manière suivante :

```
float tabx[4] = { 0.0, 1.0, 2.0, 3.0 };
float taby[]  = { 0.5, 1.5, 2.5, 3.5 };
int   tabi[100] = { 5, 5, 5, };
```

La liste des valeurs d'initialisation, placée entre accolades, peut être complète (autant de valeurs que la dimension déclarée du tableau) ou partielle.

Dans le cas d'une initialisation partielle, tableau **tabi** de l'exemple, la dernière valeur est suivie d'une virgule.

Dans le cas d'une initialisation complète, la dimension du tableau peut être omise (cas du tableau **taby** de l'exemple).

Les listes de valeurs, entre accolades, peuvent être imbriquées pour initialiser des tableaux à plusieurs dimensions :

```
double mat[2][6] = {
    { 0.0, 1.0, 2.0, 3.0, 4.0, 5.0 },
    { 0.5, 1.5, 2.5, 3.5, 4.5, 5.5 }
    };
```

NB : C n'autorise pas les initialisations de tables automatiques. On ne peut utiliser ce mécanisme que pour des tables globales, ou locales sous réserve qu'elles soient déclarées statiques.

6.3.4 Pointeurs mobiles

Ces mécanismes arithmétiques, utilisables sur pointeurs, permettent également de modifier les valeurs de pointeurs, pour leur attribuer des adresses mémoire différentes :

```
float tabx[50];
float *pf;
pf = tabx;
*pf = 0.5;  /* Acces a tabx[0]  */
pf = pf + 12;
*pf = 0.01; /* Acces a tabx[12] */
pf = pf - 2;
*pf = 1.5;  /* Acces a tabx[10] */
```

Là encore, l'unité utilisée pour l'arithmétique sur **pf** est la taille d'un **float** puisque **pf** a été déclaré comme pointeur de **float**.

Cette arithmétique vaut également pour les opérateurs d'incrémentation ou décrémentation. On trouve souvent, en C, des balayages de tableaux par pointeurs mobiles. Par exemple une initialisation :

```
float tabx[50];
float *pf;
pf = tabx + 49;
do *pf = 1.0;
while( --pf >= tabx );
```

Avec ce genre d'écritures, par ailleurs très efficaces, il faut être extrêmement précis quand à la manipulation des indices pour éviter les accès "en dehors" du tableau ! Se méfier en particulier des incrémentations ou décrémentations. La boucle ci-dessus doit s'écrire :

```
while( --pf >= tabx );
```

ou :

```
while( pf-- > tabx );
```

L'écriture :

```
while( pf-- >= tabx );
```

est un *bug*[15] qui conduira à accéder à **tabx[-1]** et donc à écraser une donnée placée en mémoire juste avant la zone allouée au tableau ou pire, à "planter" le programme sur une faute mémoire.

6.4 Structures de données

Comme tous les langages pascaliens, C permet de manipuler des données structurées. L'accès aux champs se fait par la notation symbolique classique **nom.champ**

Par exemple, des données composites de type complexe (non disponible en standard

(15) Je sais, ça semble invraisemblable que l'on puisse écrire de telles choses ! Et pourtant...

C) peuvent être implantées par structures :

```
struct complex {        /* Definition de type       */
    float real;
    float imag;
};
struct complex c1, c2; /* Declaration de variables */
c1.real = 1.0;          /* Acces aux champs         */
c1.imag = 0.0;
```

NB : ne pas confondre la définition du type structuré, qui n'est qu'une information syntaxique destinée au compilateur, avec la déclaration de variables de ce type qui crée effectivement des données.

L'affectation directe est possible entre structures de même type :

```
c2 = c1;
```

On peut, bien sûr, déclarer et utiliser des tableaux de structures en intégrant les notations "accès à un élément de table" et "accès à un champ de structure" :

```
struct complex tabc[10];
tabc[0].real = 1.0;
```

6.4.1 Alias de type

Le déclarateur **typedef**, présenté au paragraphe 3.5 page 13, est très utile lorsque l'on travaille avec des structures. La syntaxe de déclaration, un peu lourde, peut être grandement améliorée :

```
typedef struct {    /* Definition par alias     */
    float real;
    float imag;
} complex;
complex c1, c2;     /* Declaration de variables */
```

Dans cet exemple, la désignation du type structuré est bien **complex** et non plus **struct complex**.

Cette syntaxe est vivement encouragée et est utilisée implicitement dans toute la suite de ce chapitre.

6.4.2 Pointeurs de structures

Comme pour les autres types de données de C, on peut manipuler des structures par adresses et pointeurs. Les déclarations de pointeurs sont inchangées, un opérateur spécial **->** permet l'accès indirect aux champs de structures :

```
typedef struct {
    float real;
    float imag;
} complex;
complex c1, c2;    /* Variables de type "complex" */
complex *pc;       /* Pointeur sur type "complex" */
c1.real = 1.0;     /* Acces direct aux champs     */
pc = &c1;          /* Initialisation sur adresse  */
pc->real = 1.0;    /* Acces indirect a c1.real    */
```

Cet opérateur **->** sert à accéder à un champ particulier d'une structure manipulée par adresse. L'opérateur d'indirection ***** reste utilisable mais uniquement pour des opérations globales sur la structure. Par exemple une affectation :

```
c2 = *pc;
```

laquelle se lit "affecter à **c2** le contenu de **pc**".

Les pointeurs sur structures sont surtout utiles pour passer des structures en arguments d'appel de fonctions. Le passage par valeur, défaut du C, n'est pas conseillé car il peut être très lourd en cas de grosses structures. De plus, beaucoup de compilateurs le rejettent.

On construira alors une interface d'appel par pointeurs. Par exemple, une routine d'addition de complexes s'écrirait :

```
void add_complex(complex *pc1, complex *pc2)
{
    pc1->real += pc2->real;
    pc1->imag += pc2->imag;
}
```

et s'utiliserait en passant des adresses :

```
complex c1, c2;
...
add_complex(&c1, &c2);
```

6.4.3 Unions de données

L'union est une donnée structurée particulière dans laquelle les différents champs sont superposés au lieu d'être consécutifs[16].

Par exemple, on souhaite manipuler des données numériques qui peuvent être ou bien entières, ou bien réelles, mais pas les deux en même temps :

```
union data {          /* Definition de type       */
    long   i_value;
    double r_value;
};
union data d1, d2;    /* Declaration de variables */
d1.i_value = 35;      /* Interpretation en entier */
d2.r_value = 3.14159; /* Interpretation en reel   */
```

On utilisera également l'alias de type, plus lisible :

```
typedef union {
    long   i_value;
    double r_value;
    } data;
data d1, d2;
```

La taille d'une variable de type **union** est celle de son plus grand composant (et non, comme pour les structures, la somme des tailles). L'interprétation doit se faire avec rigueur. Dans cet exemple, si l'on affecte le champ entier, on devra lire le même sous peine de n'importe quoi !

(16) Mécanisme analogue au **variant record** de Pascal ou à l'**equivalence** de Fortran.

6.5 Notes commentaires

[6.1] Puisqu'une matrice 2D est en fait un vecteur de lignes, ces lignes étant elles mêmes des vecteurs de scalaires, une matrice `[M][N]` sera rangée dans l'ordre

```
[0][0], [0][1], ... [0][N-1],
```

```
[1][0], [1][1], ... [1][N-1],
```

etc.

Dans d'autres langages (Fortran) les éléments de matrice sont rangés par colonne d'abord. C'est sans grande importance SAUF lorsqu'on interface des modules écrits dans des langages différents. Par exemple, les appels depuis un programme C de routines mathématiques Fortran (librairies CERN, NAG, LAPACK, etc.) utilisant des matrices devront se faire avec précautions.

Annexe A Le préprocesseur

Le préprocesseur du C est un outil de traitement du texte source avant compilation. Il est invoqué automatiquement lorsqu'on lance une compilation (voir le paragraphe 1.1 page 1).

Il propose un ensemble de fonctionnalités qui sont mise en oeuvre à l'aide de *directives*, mots-clés spécialisés précédés du caractère **#**.

Cette annexe présente les quelques directives les plus utiles mais n'est pas exhaustive. Pour tout savoir, se reporter à la documentation du préprocesseur C :

```
man cpp
```

A.1 Inclusions de fichiers

L'inclusion d'un fichier de déclarations dans le texte source se fait par :

```
#include <nom-du-fichier>
```

ou :

```
#include "nom-du-fichier"
```

La syntaxe avec le nom du fichier entre **<..>** est réservée aux fichiers systèmes, fichiers de déclarations des fonctions de la librairie standard C. On ne spécifie que le nom du fichier, le compilateur est censé savoir dans quel répertoire le trouver[17].

La syntaxe avec le nom du fichier entre **"."** est utilisée pour les fichiers personnels, le nom est un nom simple de fichier du répertoire courant, ou un *pathname* valide.

On peut effectuer des inclusions multiples de fichiers comportant eux mêmes des directives **#include**.

A.2 Macro définitions

A.2.1 Macros simples

La directive **#define** permet de manipuler des textes sous un nom symbolique. Elle est utilisée le plus souvent pour implanter des constantes numériques sous un nom symbolique explicite, ou pour renommer des fonctions dont le nom est désagréable

(17) Répertoire **/usr/include** sur un site Unix typique

à manipuler :

```
#define PI      3.141592
#define RANDOM  math_lib_gr00a6
float x, y;
...
x = 2 * PI;
y = RANDOM(0.5);
```

NB : même si ce mécanisme est très utilisé pour définir des constantes symboliques, il ne faut pas le confondre avec le **`const`** de Pascal ou le **`parameter`** de Fortran. Il s'agit, très précisement d'une fonctionnalité de substitution, analogue au "recherche et remplace" des éditeurs de texte. On peut utiliser le **`#define`** pour de courtes séquences de code.

NB : tous les mots-clés de C ainsi que les noms des fonctions des librairies standard sont définis en minuscules. Il est habituel d'utiliser des majuscules pour les noms symboliques des macros. Ce n'est qu'une convention qui a l'avantage d'attirer l'attention lors de la lecture de sources.

L'utilisation de macros définitions permet de rendre une programmation plus souple à adapter. En particulier, on l'utilisera lorsqu'on manipule des tables :

```
#define MAXDIM  200
#define INIVAL  0.05
float tabx[MAXDIM];
int ii;
...
for( ii = 0 ; ii < MAXDIM ; ii++ ) tabx[ii] = INIVAL;
```

On pourra par la suite, et très simplement, modifier des valeurs de dimensions, d'initialisation et, en général, toutes constantes numériques plus ou moins arbitraires.

A.2.2 Suppression de macros

Une macro définition devenue inutile sera supprimée par **`#undef`**. En particulier, on ne peut redéfinir une macro qu'en supprimant d'abord la définition existante :

```
#define MAXDIM  200
...
#undef  MAXDIM
#define MAXDIM  250
...
```

A.2.3 Macros paramétrées

On peut définir des macro substitutions acceptant des arguments symboliques. Au moment de l'invocation de la macro, du texte doit être fourni pour le ou les argu-

ments symboliques. Par exemple, une macro **CUBE** s'implanterait par :

```
#define CUBE(N) N * N * N
int ii;
float x, y;
...
ii = CUBE(4);
y = 3.5;
x = CUBE(y);
```

Attention : l'exemple ci-dessus est volontairement erroné. On doit toujours garder à l'esprit que ce mécanisme est de type "recherche et remplace", le préprocesseur effectuant de la substitution de texte très stupide. Ainsi, en écrivant :

```
x = CUBE(a + b);
```

la macro développée, telle que la verra le compilateur, devient :

```
x = a + b * a + b * a + b;
```

ce qui n'est pas du tout le résultat escompté !

➤ Une régle absolue est de toujours définir les macros paramétrées en protégeant les arguments fictifs par des parenthèses. La bonne définition serait, ici :

```
#define CUBE(N)  (N) * (N) * (N)
```

Moyennant cette précaution, l'expansion de :

```
x = CUBE(a + b);
```

devient :

```
x = (a + b) * (a + b) * (a + b);
```

qui est correcte.

➤ Autre précaution, on ne doit JAMAIS utiliser des opérateurs d'incrémentation dans des macro substitutions. Une écriture anodine telle que :

```
i = CUBE(j++);
```

se développera en :

```
i = j++ * j++ * j++;
```

alors qu'on ne voulait peut-être pas incrémenter **j** trois fois !

Autre exemple, en utilisant l'opérateur de sélection **? :** (paragraphe 4.6 page 27), on peut construire des macros **MINI** et **MAXI** de deux valeurs :

```
#define MINI(A, B)  ((A) < (B)) ? (A) : (B)
#define MAXI(A, B)  ((A) > (B)) ? (A) : (B)
```

Remarque : une macro définition doit être faite sur une seule et même ligne de texte source. En cas de besoin, longues définitions, on peut indiquer la concaténation de lignes en protégeant le retour ligne par un caractère ****

```
#define MINI(A, B)  ((A) < (B)) ? \
    (A) : (B)
```

A.3 Code conditionnel

Le préprocesseur C permet d'écrire des codes source configurables moyennant certaines options. La syntaxe générale est :

```
#if <condition>
...
... code a inclure si condition vraie
#else
...
... code a inclure si condition fausse
#endif
```

Ces directives, à ne pas confondre avec la structure de contrôle **`if .. else`** du langage C, demandent au préprocesseur de traiter ou d'ignorer les lignes de code source ainsi délimitées.

La condition est évaluée selon la syntaxe du C, et accepte les mêmes opérateurs booléens. L'utilisation typique est l'écriture de sources pouvant être compilés sur des machines différentes.

Par exemple, on a signalé aux paragraphes 3.1 page 9 et 3.2 page 10 l'existence d'un type entier 64 bits, **`long long`**, non standard et inutile sur certaines machines où le **`long`** standard fait déjà 64 bits (machines DEC/Alpha).

Une application devant travailler avec des entiers 64 bits, sur différentes plateformes, utilisera un alias de type de la manière suivante :

```
#define ALPHA    0   /* 0 ou 1 selon machine     */
...
#if ALPHA
typedef long int64;
#else
typedef long long int64;
#endif
...
int64 ix, iy;           /* Definitions de variables */
```

Dans le même ordre d'idées, on peut utiliser ce mécanisme pour définir des textes spécifiques à un environnement comme, par exemple, des pathnames de fichiers :

```
#if UNIX
#define TEMPNAME    "/tmp/appli.tmp"
#endif
#if VMS
#define TEMPNAME    "SYS$SCRATCH:APPLI.TMP"
#endif
#if MSDOS
#define TEMPNAME    "C:\TEMP\APPLI.TMP"
#endif
```

A l'aide de ce mécanisme, on est donc en mesure de créer des fichiers de configuration, adaptés à différents environnements, fichiers qu'il suffit d'inclure dans tous les modules source d'une application. Les modules sources, eux, seront écrits de manière indépendante de telle ou telle plateforme.

De plus, il est possible de simuler les directives **#define** par une option de compilation :

```
cc -Dnom=valeur source.c
```

L'exemple précédent qui utilisait un :

```
#define ALPHA 0 (ou 1)
```

pourrait être traité sans placer cette directive dans le code source, et en définissant l'option au moment de la compilation :

```
cc -DALPHA=1 toto.c
```

ou :

```
cc -DALPHA=0 toto.c
```

l'avantage est que la génération du programme pour tel ou tel environnement ne nécessite plus de modifier le texte source ! Les différentes options de génération peuvent être implantées via un fichier *makefile*.

A.3.1 Macros prédéfinies

La directive **#if** exploite une expression booléenne qui peut être construite à partir de constantes symboliques. Il est possible de se contenter de tester l'existence ou la non existence d'une macro sans se préoccuper de sa valeur :

```
#define OPTION           /* Definition sans valeur */
#if defined(OPTION)      /* Test existence         */
...
#if ! defined(OPTION)    /* Test non existence     */
```

Ces types de conditions sont tellement courantes qu'il existe des directives spécialisées :

```
#ifdef OPTION        /* Test existence       */
#ifndef OPTION       /* Test non existence   */
```

Enfin, tous les environnements C prédéfinissent un certain nombre de macros, qui sont automatiquement créées lorsqu'on lance une compilation, et qu'on pourra donc interroger directement. Par exemple :

```
#ifdef __unix /* On compile sur une machine Unix  */
...
#ifdef __vms  /* On compile sur une machine VMS   */
```

La liste de ces macros prédéfinies dépend des compilateurs. Se reporter aux documentations :

```
man cc
```

Annexe B Les déclarateurs de type

Cette annexe récapitule les déclarateurs de types de données de C, présentées au chapitre 3 page 9, et indique - quand il existe - l'équivalent syntaxique Fortran.

B.1 Types scalaires et attributs

Type donnée	Syntaxe C	Syntaxe F77
Entier par défaut	**`int`**	**`integer`**
Entier 8 bits	**`char`**	**`byte`**
Entier 16 bits	**`short`**	**`integer*2`**
Entier 32 bits	**`int`** ou **`long`**	**`integer*4`**
Entier 64 bits	**`long`** ou **`long long`**[18]	**`integer*8`**[18]
Booléen	(implicite numérique)	**`logical`**
Caractère	(implicite numérique, **`char`** ou **`int`**)	**`char`**
Chaîne	(tableau de **`char`**)	**`character*`***n*
Réel simple	**`float`**	**`real*4`**
Réel double	**`double`**	**`real*8`**
Complexe	—	**`complex`**
Attribut représentation non signée	**`unsigned`**	—
Attribut *read-only*	**`const`**[19]	—
Attribut fugitif	**`volatile`**[19]	—

(18) Si disponible, selon machine.

(19) C Ansi uniquement.

Annexe C Les opérateurs et symboles

Cette annexe récapitule les opérateurs du C, présentés au chapitre 4 page 19, et indique - quand il existe - l'équivalent syntaxique Fortran.

C.1 Opérateurs arithmétiques

Nature opération	Syntaxe C	Syntaxe F77
Addition	`+`	`+`
Soustraction ou négation	`-`	`-`
Multiplication	`*`	`*`
Division	`/`	`/`
Modulo	`%`	`mod`
Puissance	—	`**`
AND binaire	`&`	`iand`[20]
OR binaire	`|`	`ior`[20]
XOR binaire	`^`	`ixor`[20]
Complément binaire	`~`	—
Décalage de bits	`<<` `>>`	`ishft`[20]

C.2 Comparateurs

Nature opération	Syntaxe C	Syntaxe F77
Inférieur	`<`	`.lt.`
Inférieur ou égal	`<=`	`.le.`
Supérieur	`>`	`.gt.`
Supérieur ou égal	`>=`	`.ge.`
Égal	`==`	`.eq.`
Différent	`!=`	`.ne.`

C.3 Opérateurs booléens

Nature opération	Syntaxe C	Syntaxe F77
ET booléen	`&&`	`.and.`
OU booléen	`||`	`.or.`
NON booléen	`!`	`.not.`
"Vrai" booléen	arithmétique non nul	`.true.`
"Faux" booléen	arithmétique nul	`.false.`

(20) Fonction arithmétique, disponible avec certains Fortran seulement.

C.4 Divers

Nature opération	Syntaxe C	Syntaxe F77
Indice tableau	`[..]`	`(..)`
Indices multiples	`[..][..]`	`(..,..)`
Membre de structure	`.`	`.`
Pointeur membre	`->`	—
Adresse	`&`	—
Indirection	`*`	—

C.5 Evaluation des opérations

Ce tableau présente les niveaux de priorité d'évaluation des opérateurs de C, de la priorité maximale à la priorité minimale, ainsi que l'associativité, gauche à droite ou droite à gauche, pour l'ordre d'évaluation.

Priorité	Opérateurs	Associativité
maxi	`[] () . ->`	▷ ▷
▽	`-`[21] `~ ! *`[22] `&`[23] `++ -- sizeof cast`	◀ ◀
▽	`*`[24] `/ %`	◀ ◀
▽	`+ -`[25]	▷ ▷
▽	`<< >>`	▷ ▷
▽	`< <= > >=`	▷ ▷
▽	`== !=`	▷ ▷
▽	`&`[26]	▷ ▷
▽	`^`	▷ ▷
▽	`|`	▷ ▷
▽	`&&`	▷ ▷
▽	`||`	▷ ▷
▽	`? :`	◀ ◀
▽	`= *= /= %= += -= <<= >>= &= |= ^= ~=`	◀ ◀
mini	`,`[27]	▷ ▷

(21) Négation
(22) Indirection
(23) Adresse
(24) Multiplication
(25) Soustraction
(26) AND binaire
(27) Séquenceur

Annexe D Caractères et chaînes

Contrairement à d'autres langages informatiques où les chaînes de caractères font l'objet d'une implantation explicite, **`string`** en Pascal, **`character*`***n* en Fortran, en C les chaînes de caractères ne sont pas supportées directement.

On devra donc les implanter explicitement, par l'intermédiaire de tableaux et pointeurs d'octets, et les manipuler par des fonctions spécialisées, développées pour l'occasion ou appelées dans les librairies standard.

D.1 Conventions d'implantation

Comme il a été signalé au paragraphe 3.1 page 9, le caractère de texte n'est pas un type explicite du langage. On le manipule par l'intermédiaire d'un type numérique qui est son encodage Ascii. Le type numérique **`char`**, entier 8 bits, est le type privilégié pour des raisons d'économie mémoire.

La traduction caractère en son code Ascii est effectuée par le compilateur via la notation symbolique `' '` dite "constante caractère". Par exemple, **`'A'`** désigne le code Ascii **`65`**.

Une chaîne lexicale, en C, sera donc un tableau de **`char`**, de taille ad-hoc. Par convention, le code Ascii nul est utilisé comme terminateur de chaîne :

```
char chaine[20];
chaine[0] = 'A';
chaine[1] = 'B';
chaine[2] = 0;        /* Terminateur  */
```

Une seconde syntaxe permet de déclarer une chaîne initialisée :

```
char *message = "Hello";
```

La notation `"..."`, dite "constante chaîne" est un raccourci équivalent à l'initialisation d'un tableau d'octets :

```
char message[] = { 'H', 'e', 'l', 'l', 'o', 0 };
```

NB : on notera qu'avec cette écriture, l'octet nul (terminateur conventionnel) est prévu.

D.2 Caractères spéciaux

La notation constante caractère permet également de désigner des codes de contrôle et autres caractères spéciaux :

`\a` Caractère de contrôle "alerte" (bip sonore, ou **`Ctrl-G`**).

`\b` Caractère de contrôle "backspace".

\f	Caractère de contrôle "effacement" (Form Feed, ou **Ctrl-L**).
\n	Caractère de contrôle "fin de ligne".
\r	Caractère de contrôle "carriage return".
\t	Caractère de contrôle "tabulation".
nnn	Tout caractère de code octal *nnn*. Par exemple le caractère d'échappement, "Escape", se notera **\033**
\x*nn*	Tout caractère de code hexadécimal *nn*. Par exemple le caractère d'échappement, "Escape", se notera **\x1B**

Ces notations sont utilisables en définition de caractères :

```
char new_line[2] = { '\n', 0 };
```

ou dans des constantes chaînes :

```
char *clear_vt100 = "\033[2J\033[H";
```

➤ Le caractère symbolique **\n** joue un rôle à part et ne correspond pas nécessairement à un caractère de contrôle précis. Lorsque l'on manipule des fichiers texte en particulier, le délimiteur de lignes est spécifique à tel ou tel environnement : ce sera le code contrôle "Line Feed", Ascii 10, dans les environnements Unix, le code "Carriage Return", Ascii 13, dans l'environnement Apple MacIntosh, la séquence "Carriage Return + Line Feed", Ascii 13,10, dans l'environnement Ibm MsDos, etc.

Cette notation **\n** signifie donc "code fin de ligne en vigueur sur cette machine", et son utilisation systématique est conseillée pour assurer la portabilité des programmations.

D.3 Fonctions de manipulation

Toutes les manipulations de chaînes de caractères, en C, passent par des fonctions spécialisées qui exploitent le fait qu'une chaîne est implantée via un pointeur d'octets et la convention "le terminateur de chaîne est l'octet nul". Pour illustrer ce principe, on montre deux exemples de fonctions classiques.

Calcul de la longueur d'une chaîne, i.e. comptage des octets pointés jusqu'à trouver un octet nul :

```
int strlen(const char *str)
{
    int size = 0;
    while( *str++ ) size++;
    return size;
}
```

Copie d'une chaîne, i.e. transfert des octets d'un pointeur origine vers un pointeur destination, jusqu'à trouver un octet nul. La chaîne destination devra être terminée aussi par un octet nul :

```
void strcpy(char *dst, const char *org)
{
    while( *org ) *dst++ = *org++;
    *dst = 0;
}
```

Ces fonctions, et beaucoup d'autres, existent déjà dans la librairie du C et n'ont donc pas besoin d'être reprogrammées. Il est tout de même intéressant de connaître leur principe afin de les utiliser correctement. En effet, certaines de ces fonctions sont potentiellement dangereuses, c'est le cas de **strcpy** qui recopie des octets, dans une boucle, jusqu'à trouver la fin conventionnelle d'une chaîne origine. On peut observer que la place mémoire disponible, à partir du pointeur destination, n'est pas gérée !

On s'expose donc à des déboires en utilisant cette fonction sans précautions :

```
char message[10];
strcpy(message, "Coucou");                  /* Correct  */
...
strcpy(message, "Message assez long..."); /* Erreur ! */
```

Dans le second cas, on va copier 21 octets plus un octet nul dans un tableau dimentionné à 10, provoquant un écrasement mémoire !

D.4 Les fonctions standard

D.4.1 Fonctions chaînes

La librairie standard de C dispose d'un ensemble de fonctions de manipulation de chaînes. Les prototypes de ces fonctions figurent dans un fichier en-tête, à inclure dans les modules source :

```
#include <string.h>
```

Les fonctions les plus courantes[28] sont :

int strlen(const char* s);
: Retourne la longueur effective de la chaîne pointée par **s**, en octets (implantation illustrée au paragraphe précédent).

char* strcpy(char *s1, const char *s2);
: Copie la chaîne pointée par **s2** à l'adresse **s1** (implantation illustrée au paragraphe précédent).

char* strncpy(char *s1, const char *s2, int n);
: Analogue à la précédente, mais copie au maximum **n** octets si un code nul n'est pas trouvé avant. C'est la variante "sécurisée" de **strcpy** qui évite le problème d'écrasement évoqué au paragraphe précédent !

char* strcat(char *s1, const char *s2);
: Concatène la chaîne pointée par **s2** à **s1**. Cette fonction transfère les octets de **s2**, jusqu'à octet nul, à partir du premier octet nul trouvé en aval de **s1**.

int strcmp(const char *s1, const char *s2);
: Compare les chaînes pointées par **s1** et **s2**, et retourne une valeur nulle si les chaînes sont identiques (même longueur jusqu'à octet nul et mêmes octets).

  Cette fonction retourne une valeur négative si **s1** est "avant" **s2**, au sens lexicographique habituel (ordre alphabétique), ou retourne une valeur positive si **s1** est "après" **s2**.

(28) Liste non exhaustive

int strncmp(const char *s1, const char *s2, int n);
Analogue à la précédente, comparaison limitée aux **n** premiers caractères.

D.4.2 Fonctions caractères

La librairie standard de C dispose également d'outils de manipulation de caractères isolés. On devra inclure le fichier en-tête :

```
#include <ctype.h>
```

On trouvera des outils tels que :

int isalpha(int c);
Retourne un résultat booléen vrai (non nul) si le code caractère argument est alphabétique.

int isupper(int c)
Retournent un résultat booléen vrai (non nul) si le code caractère argument est alphabétique majuscule, ou minuscule.

int isdigit(int c);
Retourne un résultat booléen vrai (non nul) si le code caractère argument est un chiffre.

int toupper(int c);
Retourne le code caractère résultat de la conversion en majuscule du code caractère argument (qui doit être alphabétique).

int tolower(int c);
Analogue, conversion en minuscule.

NB : Les fonctions ci-dessus sont implantées en utilisant l'alphabet ASCII. Ne pas perdre de vue que l'informatique est une activité anglo-saxonne. Lorsqu'on développe des programmes manipulant du texte utilisant les alphabets européens ISO, ces fonctions seront rarement utilisables. Par exemple la fonction **toupper** de conversion en majuscule traduira correctement **e** en **E** mais pas **é** en **É**.

D.4.3 Compléments

Tous les outils disponibles en librairie ne sont que des "basics". On peut toujours construire des compléments adaptés à tel ou tel besoin.

Par exemple, conversion en majuscules d'une chaîne ASCII :

```
#include <ctype.h>
void strupper(char *str)
{
    while( *str ) {
       if( isalpha(*str) ) *str = toupper(*str);
       str++;
    }
}
```

Ou encore, conversion d'une chaîne au standard Fortran (taille fixe, chaîne terminée

par un remplissage d'espaces) vers un tableau d'octets C :

```
#include <string.h>

void cstr_fstr_cpy(char *cs, int cn, char *fs, int fn)
{
    /* Ignore les espaces terminaux */
    while( fn && (*(fs + fn - 1) == ' ') ) fn--;

    /* Controle/troncature */
    if( fn >= cn ) fn = cn - 1;

    /* Copie et octet nul final */
    if( fn ) strncpy(cs, fs, fn);
    cs[fn] = 0;
}
```

Annexe E Les entrées/sorties

Dans la librairie standard de C, les fonctions les plus utilisées sont certainement les "entrées/sorties standard", assurant tous les accès fichiers texte et binaires, et les lectures et écritures au terminal.

Contrairement à d'autres systèmes d'exploitation, Unix ne supporte qu'un seul format physique de fichiers, le *byte stream*. Toutes les fonctions d'entrées sorties et de positionnement sont orientées octets. Les fichiers de type **record**, **indexed**, etc. n'existent pas.

Les prototypes des fonctions de la librairie standard d'entrées/sorties se déclarent par inclusion dans les modules sources d'un fichier spécifique :

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

E.1 Fichiers texte

E.1.1 Ouverture

Un fichier, en C, est manipulé par l'intermédiaire d'un pointeur vers une structure gérée par la librairie C, et nommée **FILE**.

Le code application devra déclarer un pointeur de ce type, et l'initialiser (lui donner une valeur valide) par un appel d'ouverture :

```
FILE* fp;
fp = fopen(<name>, <mode>);
```

L'argument **<name>** est un pointeur caractères spécifiant le nom du fichier, nom local ou pathname complet. L'argument **<mode>** est un pointeur caractères définissant le mode d'ouverture du fichier.

Par exemple :

```
fp = fopen("/import/projet/toto.data", "r+");
```

Les modes disponibles sont :

"r" Ouverture en lecture seule, le fichier doit exister.

"r+" Ouverture en lecture et écriture, le fichier doit exister.

"w" Ouverture en écriture seule. Le fichier peut ne pas exister auquel cas il est créé. Si le fichier existe déjà il est réinitialisé et vidé !

"w+" Ouverture en écriture et lecture. Le fichier est créé s'il n'existe pas ou vidé s'il existe déjà.

"a" Ouverture en écriture et en mode **append**. La logique de traitement du fichier est la même que pour le mode **w**, mais toutes les écritures sont ajoutées en fin du fichier, l'accès aléatoire n'est pas possible.

"a+" Ouverture en lecture et écriture, et en mode **append** pour les écritures.

La fonction **fopen**, de type **FILE***, retourne un pointeur valide après une ouverture réussie. En cas de problème, et par convention, cette fonction retourne un pointeur nul. On devra donc toujours tester la valeur retournée :

```
fp = fopen(..........);
if( fp == (FILE*)0 ) ... Erreur
else ... Correct
```

Le pointeur obtenu sera ensuite passé en argument à tous les appels ultérieurs, relatifs à ce fichier.

En particulier, la fermeture se fera par :

```
fclose(fp);
```

NB : après fermeture, la valeur du pointeur n'a plus aucun sens et ne doit plus être utilisée.

E.1.2 Sorties en mode texte

La librairie C propose trois fonctions d'écriture texte :

int fputc(int c, FILE *F);

Écriture d'un caractère **c** vers le fichier **F**. Retourne le caractère argument, ou **-1** en cas d'erreur d'écriture.

int fputs(const char *s, FILE *F);

Écriture d'une chaîne **s**. Retourne le nombre d'octets écrits, ou **-1** en cas d'erreur.

int fprintf(FILE *F, const char *format, ...);

Écriture formatée d'une liste d'arguments. Le format est décrit par un pointeur chaîne, les arguments, séparés par des virgules, sont quelconques en nature et en nombre, sous réserve de cohérence avec le format.

E.1.3 Formats d'entrées/sorties

En C, les formats sont implantés par une chaîne de caractères comportant du texte libre (à copier tel quel) et des descripteurs de champs. Ces descripteurs sont repérés par le caractère **%** et ont la structure suivante :

```
%<justif><taille><.precis><long><type>
```

- **<type>** est une information obligatoire spécifiant la nature de la donnée à afficher (nature de l'argument), sous forme d'une lettre clé :

  d Argument numérique entier (**int**).

  u Argument numérique non signé (**unsigned**).

  x (ou **X**), argument numérique entier, formatage en hexadécimal minuscule ou majuscule.

  o Argument numérique entier, formatage en octal.

  f Argument réel, formatage en virgule fixe.

  e Argument réel, formatage en notation exponentielle.

  c Argument numérique entier, formatage caractère (par exemple af-

fichage de **A** pour un argument de valeur **65**).

s Argument pointeur de chaîne de caractères.

- **<long>**, élément optionnel, est la lettre **l** ajoutée au code <type> lorsque l'argument est un entier numérique long, ou un réel double précision. On peut donc avoir des spécificateurs de type complets tels que **ld**, **lu**, **le**, etc.
- **<taille>**, élément optionnel, spécifie la taille totale du champ formaté. Par exemple **%5d** indique un formatage numérique entier dans un champ de 5 caractères. La justification est à droite par défaut, le champ est complété par des espaces ajoutés en tête.

  Si l'on souhaite afficher avec des **0** initiaux, on fera figurer un **0** dans la spécification de taille, par exemple **%05d**

  Lors du formatage de chaînes (type **s**), la justification est également à droite par défaut.
- **<.precis>**, élément optionnel, spécifie le nombre de décimales pour les affichages de réels. Par exemple **%.4f** indique un format en virgule fixe, avec quatre décimales.

  On peut cumuler ce spécificateur avec la taille, **%8.4f** indique un format en virgule fixe, quatre décimales, dans un champ de taille totale huit caractères (y compris le signe et le point décimal).
- **<justif>**, élément optionnel. Lorsqu'il est présent, ce doit être un signe **-** demandant d'inverser la justification par défaut (à droite pour les numériques et chaînes).

E.1.4 Exemples de formats

```
char *name = "xmin";
int xmin = 0, xmax = 255;
fprintf(.., "Bornes : %s = %4d, %s = %4d\n",
            name, xmin, "xmax", xmax);
```

donnera le résultat :

```
Bornes : xmin =    0, xmax =  255
```

Autre exemple :

```
int car = 65;
fprintf(.., "Le code ascii de %c est %d !\n", car, car);
```

donnera le résultat :

```
Le code ascii de A est 65 !
```

(On remarquera la distinction entre les types formats **c** et **d**)

NB : contraitement à Fortran, les passages à la ligne doivent être explicites. On notera la présence, dans la partie "texte libre" du format, du pseudo code **\n**, caractère de contrôle fin de ligne.

NB : en C, les formats ne sont pas impératifs. En particulier il n'y a jamais de débordements si un spécificateur de format est "trop petit". La règle est d'afficher l'argument, dans tous les cas, quitte à bousculer le colonnage, et non, comme le fait Fortran, de remplir le champ avec des caractères *****.

NB : on peut utiliser la fonction **fprintf** avec un descripteur de format simple,

texte libre sans spécificateurs. Dans ce cas, on ne passe aucun autre argument :

```
fprintf(.., "Hello world\n");
```

le résultat est équivalent à l'utilisation de **fputs**.

E.1.5 Entrées en mode texte

La librairie C propose trois fonctions de lecture texte :

int fgetc(FILE *F);

Lecture d'un caractère, depuis le fichier **F**. Retourne le code caractère ou **-1** en cas d'erreur ou de fin de fichier.

char *fgets(char *s, int len, FILE *F);

Lecture d'une ligne de texte, depuis le fichier **F**, dans un tableau d'octets pointé par **s**. La lecture est faite jusqu'à rencontrer un code fin de ligne ou jusqu'à ce que la taille maximale spécifiée par l'argument **len** soit atteinte. Retourne le pointeur argument **s** ou un pointeur nul en cas d'erreur.

Cette fonction ne copie pas d'octet nul en fin de chaîne. Pour exploiter le texte lu à partir de fonctions de la librairie chaînes (cf. annexe D page 55), on devra le prévoir :

```
FILE *fp;
char line[80];
if( fgets(line, 79, fp) == (char*)0 ) ... Erreur
line[79] = 0;   /* Terminateur chaine  */
```

int fscanf(FILE *F, const char *format, ...);

Lecture formatée. L'argument **format** est une chaîne comportant des spécificateurs de format, identiques à ceux utilisés pour les écritures. Les arguments doivent être des adresses de variables du type convenable compte tenu des spécificateurs.

Par exemple, lecture d'une ligne de texte comportant deux valeurs réelles séparées par une virgule :

```
FILE* fp;
float x, y;
fscanf(F, "%f,%f\n", &x, &y);
```

Cette fonction retourne le nombre de conversions réussies, 2 dans cet exemple.

E.1.6 Entrées/sorties au terminal

En parallèle de ces fonctions très générales, la librairie d'entrées/sorties de C dispose de trois pointeurs fichiers, déjà initialisés :

stdin Entrée standard, fichier en lecture seule initialisé vers le terminal (clavier), jouant un rôle analogue à l'unité **5** de Fortran.

stdout Sortie standard, fichier en écriture seule initialisé vers le terminal (écran), jouant un rôle analogue à l'unité **6** de Fortran.

stderr Sortie erreurs, fichier en écriture seule initialisé vers le terminal (écran).

On peut utiliser ces descripteurs directement, i.e. sans devoir faire un **fopen**. Le cas courant est l'envoi de messages sur la sortie erreurs :

```
FILE *fp;
char *name = "toto.data";

fp = fopen(name, "r");
if( fp == (FILE*)0 ) {
    fprintf(stderr, "Fichier %s invalide\n", name);
    ...
}
```

D'autre part, toutes les fonctions d'entrées/sorties texte existent sous forme d'une variante spécifique terminal, portant le même nom au caractère initial **f** près. Ainsi :

```
printf(format, ...);
```

est l'équivalent de :

```
fprintf(stdout, format, ...);
```

ou encore :

```
car = getchar();
```

est l'équivalent de :

```
car = fgetc(stdin);
```

etc.

La seule exception concerne la lecture chaîne **fgets**. La variante **gets** est ultra simplifiée :

```
char *gets(char *s);
```

et ne comporte plus l'argument spécifiant une taille maximale. On ne doit donc utiliser cet appel qu'avec un buffer d'octets de taille suffisante, pour éviter un écrasement.

E.2 Fichiers binaires

Les fichiers binaires permettent des entrées/sorties de données au format interne de la machine. Les accès sont considérablement plus efficaces que pour des entrées/sorties formatées, par contre les fichiers ne sont plus portables entre machines différentes.

E.2.1 Ouverture

L'ouverture d'un fichier binaire se fait en ajoutant le spécificateur **b** au mode d'ouverture de l'appel **fopen**, toutes les autres options restant utilisables. Par exemple :

```
FILE *fi, *fo;
fi = fopen("data.bin", "r+b");
fo = fopen("result.bin", "wb");
```

E.2.2 Entrées/sorties

Les accès se font via deux fonctions, **fread** et **fwrite**, avec une interface banalisée : adresse de donnée, taille de la donnée, nombre de données (dans le cas d'utilisation de tableaux) et, bien sûr, le pointeur fichier.

Pour les tailles des données à lire ou écrire, il est conseillé d'utiliser systématiquement l'opérateur **sizeof**.

Exemples :

```
double x, y;
float tab[50];
/* Lecture sur 'fi' de 2 scalaires */
fread(&x, sizeof(double), 1, fi);
fread(&y, sizeof(double), 1, fi);
/* Ecriture sur 'fo' d'un tableau */
fwrite(tab, sizeof(float), 50, fo);
```

Ces fonctions retournent le nombre de données lues ou écrites, qui doit être égal au troisième argument de l'appel sauf problème.

Remarque : certaines des fonctions de lecture texte sont utilisables sur un fichier ouvert en mode binaire. En particulier les fonctions **fgetc** et **fputc** permettent une lecture ou écriture binaire, octet par octet.

E.3 Divers

int feof(FILE *F);

Retourne une expression logique (au sens de C, nulle ou non nulle) lorsque la lecture est arrivée en fin de fichier. Cet état ne peut être testé qu'après une lecture !

```
FILE *fp;
char line[80];
fgets(line, 79, fp);
if( feof(fp) ) ... C'est fini
```

Remarque : lors de l'utilisation d'une lecture texte, caractère par caractère, par la fonction **fgetc**, la détection de fin de fichier peut aussi se faire par examen du code caractère retourné. C'est un entier positif, code compris entre **0** et **255** et donc, par convention, la fonction **fgetc** retourne une constante numérique symbolique **EOF** négative lorsque le fichier est épuisé.

long ftell(FILE *F);

Retourne, sous forme d'un entier long, la position courante de lecture ou écriture du fichier **F**.

Remarque : un fichier *byte stream* Unix ne dispose que d'un seul index pour les lectures et écritures. Cet index est mis à jour après chaque entrée/sortie, et pointe la prochaine position.

int fseek(FILE *F, long pos, int mode);

Change la position courante de lecture ou écriture du fichier **F**. L'argument **pos**, entier long, spécifie un décalage, l'argument **mode** indique l'interprétation du décalage. Cet argument peut prendre une des valeurs suivantes (définies sous forme de constantes symboliques dans le fichier **stdio.h**) :

SEEK_SET Le décalage **pos** s'applique à partir du début du fichier.

SEEK_CUR Le décalage **pos** est relatif à la position actuelle de lecture/écriture.

SEEK_END Le décalage **pos** est relatif à la fin du fichier.

Cette fonction **fseek** retourne **0** en cas de positionnement réussi, un code erreur non nul sinon.

Exemples de positionnements :

- Avance en fin de fichier :

```
fseek(fp, 0, SEEK_END);
```

- Retour en début de fichier :

```
fseek(fp, 0, SEEK_SET);
```

- "Saute" deux réels :

```
fseek(fp, 2 * sizeof(float), SEEK_CUR);
```

- etc.

NB : on ne dispose pas, en C, de fonction donnant la taille d'un fichier en octets. Cette information est triviale à obtenir, il suffit d'aller en fin du fichier et de lire la position courante :

```
FILE *fp;
long filesize;

fseek(fp, 0, SEEK_END);
filesize = ftell(fp);
```

(et de ne pas oublier de revenir au début si l'on souhaite ensuite effectuer des lectures !)

Annexe F La librairie mathématique

Contrairement à Fortran, C n'est pas un langage a priori orienté calcul scientifique[29]. De ce fait, la notion de fonction intrinsèque n'existe pas.

L'utilisation de fonctions mathématiques dans des codes C passera par la déclaration des prototypes, via l'inclusion d'un fichier standard :

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

et nécessitera une édition de liens explicite avec la librairie mathématique, voir le paragraphe 1.2 page 2.

Les fonctions de la librairie mathématique C ne sont disponibles qu'en version double précision, de fait tous les prototypes ont l'allure suivante :

```
double sin(double arg);
```

On indique ci-après les noms syntaxiques des fonctions mathématiques. Cette liste n'est pas exhaustive, les implantations de librairies mathématiques pouvant varier d'une plateforme à l'autre :

- Fonctions circulaires : **sin**, **cos**, **tan**.
- Fonctions circulaires inverses : **asin**, **acos**, **atan**. Une variante de **atan**, **atan2(x,y)**, évalue l'**Arctg** de **x/y** en traitant les cas **y = 0** et les signes.
- Fonctions hyperboliques : **sinh**, **cosh**, **tanh**. Les fonctions hyperboliques inverses ne sont, sauf exception, pas disponibles. En cas de besoin on les évaluera par des formules d'équivalence.
- Fonctions néperiennes : **exp**, **log**, **log10**. La fonction générique **pow(x,y)** évalue **x^y** et permet, en particulier, de calculer **10^y**.
- Fonctions arithmétiques : on dispose de la racine carrée **sqrt**, de la valeur absolue **fabs**, du modulo **fmod** (fonction à deux arguments **fmod(x, y)**), et des troncatures entières plancher **floor** et plafond **ceil**.
- Fonctions diverses : certaines implantations proposent les fonctions des erreurs, directe et complémentaire, **erf** et **erfc**, et la fonction Γ. Se méfier des documentations approximatives[30], certaines librairies mathématiques documentent une fonction Γ, mise en oeuvre sous la forme **gamma(x)** qui retourne en réalité la valeur $log(\Gamma(x))$! D'autres librairies proposent deux fonctions, **gamma** et **lgamma**.

  Enfin, on trouvera dans certaines implantations les fonctions de Bessel, de première et deuxième espèce, pour les ordres **0**, **1** et **n**.

(29) Ne pas oublier que C a été créé initialement pour écrire le système d'exploitation Unix.

(30) Mais si, ça existe !

F.1 Constantes

Enfin, le fichier en-tête **`math.h`** implante également quelques définitions symboliques pour les constantes usuelles : **`M_PI`**, **`M_E`**, **`M_LN2`**, **`M_LN10`**, etc.

En cas de besoin, une consultation de ce fichier (les fichiers en-tête de la librairie standard sont habituellement dans le répertoire **`/usr/include`**) peut s'avérer profitable.

F.2 Précautions

Comme indiqué ci-dessus, C n'est pas un langage orienté calcul scientifique comme peut l'être Fortran. Les outils existent, on peut donc écrire des codes de calcul sans problèmes particuliers. Les contrôles sont minimalistes, voire inexistants.

Ainsi sur un certain nombre de plateformes, des appels de fonctions erronés tels que **`sqrt(x)`** ou **`log(x)`** avec un argument négatif retourneront simplement une valeur nulle, sans provoquer d'erreur ou d'interruption d'exécution.

On peut donc tout à fait lancer des programmes qui s'exécuteront jusqu'au bout, sans problèmes, donnant des résultats faux. Il appartient au programmeur de faire preuve de vigilance dans les développements.

Annexe G L'interface système Unix

Le langage C a été créé, initialement, pour écrire le système d'exploitation Unix. Il n'est donc pas hautement surprenant de trouver, dans les librairies standard, une pléthore d'appels systèmes et de fonctionnalités liées à Unix.

La liste de ces outils déborde du cadre de ce guide[31]. On va simplement évoquer deux mécanismes d'emploi courant.

Les fonctions de base de l'interface C Unix sont prototypées dans un fichier en-tête, à inclure systématiquement dès que l'on exploite ces outils :

```
#include <stdlib.h>
```

G.1 Interface processus

Un processus Unix est lancé par une commande, laquelle comporte le nom du programme et, éventuellement, des options et arguments. Par exemple :

```
cc -c -O2 toto.c
```

Le programme **cc**, au démarrage, est censé récupérer les options **-c** et **-O2** ainsi que l'argument nom du fichier **toto.c**.

Pour ce faire, l'interpréteur de commandes[32] "découpe" la ligne de commande en utilisant les espaces de séparation entre les mots et construit une table de pointeurs. Le programme est ensuite lancé, par appel de sa fonction **main()** (cf. paragraphe 3.9 page 17), en lui passant en arguments les informations de la ligne de commande :

```
int main(int argc, char **argv)
{
   ...
}
```

Le premier argument, **argc** (*args count*), indique le nombre de mots de la ligne de commande. Le second, **argv** (*args values*), est un pointeur de pointeurs de **char**, donc une table de pointeurs de **char**, dont les "**argc**" éléments pointent vers les mots de la ligne de commande.

Dans l'exemple évoqué ci-dessus, commande :

```
cc -c -O2 toto.c
```

l'argument **argc** vaudrait 4, et les pointeurs **argv** pointeraient vers :

(31) Le manuel de référence de programmation Unix dépasse les 1000 pages !

(32) Le *shell*.

```
argv[0]   ▷  "cc"
argv[1]   ▷  "-c"
argv[2]   ▷  "-O2"
argv[3]   ▷  "toto.c"
```

Ce mécanisme permet donc au programme lancé de récupérer (et traiter) toute la ligne de commande. Le traitement est, bien sûr, entièrement à la charge du programme lui-même.

NB : le nom du programme lui-même est toujours le premier argument de la ligne de commande. De fait, l'argument d'appel **`argc`** vaut toujours au minimum 1.

G.1.1 Retour d'exécution

Lorsqu'un processus Unix s'arrête, il est censé retourner au système un compte rendu numérique. Par convention, celui-ci est nul lorsque tout s'est bien passé, un code retour non nul signale un problème.

Pour terminer l'exécution d'un programme C en retournant un compte rendu au système, on disposera de deux moyens :

1. Quitter la fonction **`main`** en retournant une valeur numérique nulle ou non nulle :

```
int main(int argc, char **argv)
{
   ...
   if( probleme ) return 1;
   ...
   return 0;    /* Fin d'execution Ok */
}
```

2. Appeler, à tout moment et même depuis une fonction du programme, la fonction de sortie :

```
exit( valeur );
```

   Cette fonction arrête l'exécution et retourne au système le compte rendu **`valeur`** passé en argument.

G.2 Gestion mémoire

C supporte une gestion dynamique de mémoire.

L'allocation statique de mémoire, pour des données de taille importante : tables, matrices, etc., passe par une déclaration lors de l'écriture du code source. Par exemple :

```
float tabsta[150];
```

le compilateur se chargeant alors de réserver la zone mémoire et d'initialiser l'identificateur de table (pointeur) sur cette zone. On accède ensuite aux données par indexation **`tabsta[i]`**.

L'allocation dynamique permet de gérer des tables dont la taille ne sera connue qu'en exécution. Le code application doit simplement déclarer un pointeur, de type ad-hoc, et, en exécution, demander de la mémoire au système et affecter au pointeur l'adresse

retournée.

La taille à allouer doit être calculée par l'application et est toujours spécifiée en octets. On peut "agrandir" une zone déjà allouée, si besoin est. Lorsque l'application n'a plus besoin d'une zone mémoire, on doit la "rendre" au système.

Exemple de mise en oeuvre des trois appels de base :

```
float *tabdyn;    /* Declaration pointeur */
...
/* Allocation pour 1000 reels */
tabdyn = malloc(1000 * sizeof(float));

/* Utilisation ... */
tabdyn[0] = 0.5;

/* Reallocation pour 1500 */
tabdyn = realloc(tabdyn, 1500 * sizeof(float));

/* Liberation de la zone */
free(tabdyn);
```

➤ La mémoire machine n'est pas une ressource inépuisable et le système n'est jamais tenu de répondre à toute requête d'allocation. Par convention, les fonctions d'allocation **malloc** et **realloc** retournent un pointeur nul (adresse invalide) si la requête est rejetée. Une application doit TOUJOURS contrôler le résultat d'une allocation avant d'utiliser la zone mémoire en question :

```
float *tabdyn;

tabdyn = malloc(10000 * sizeof(float));
if( tabdyn == (float*)0 ) {
   fprintf(stderr, "On est mal !\n");
   exit(1);
}
...
```

Index

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W