DEA Astrophysique et milieux dilués

Cours de Langage C

—. — .—

Notions de programmation structurée

Outils de développement

A. Pécontal

Arlette.Pecontal@obs.univ-lyon1.fr

Centre de Recherche Astronomique de Lyon
9, Avenue Charles André
F-69561 S^t Genis Laval Cedex

14 novembre 2002

Table des matières

1 Bases de programmation **5**
- 1.1 Langage de programmation . . . 5
- 1.2 Compilation et édition de lien . . . 5
  - 1.2.1 Compilation . . . 5
  - 1.2.2 Édition de lien . . . 6
  - 1.2.3 Avantage et inconvénient des langages compilés : . . . 7
- 1.3 Programmation structurée . . . 7
  - 1.3.1 Modularité des programmes . . . 7
  - 1.3.2 Bibliothèques . . . 7
- 1.4 Documentation - Lisibilité des sources . . . 7

2 Langage C **9**
- 2.1 Règles de syntaxe du C . . . 9
- 2.2 Les types . . . 9
- 2.3 Les opérations simples . . . 10
  - 2.3.1 L'affectation . . . 10
  - 2.3.2 Les opérateurs arithmétiques . . . 10
  - 2.3.3 Les raccourcis . . . 10
  - 2.3.4 Les opérateurs de traitement de bits . . . 11
  - 2.3.5 Les opérateurs de comparaison . . . 11
  - 2.3.6 L' opérateur & - Notion d'adresse . . . 11
- 2.4 Les conversion de type . . . 11
- 2.5 Programme principal et appel de fonctions . . . 12
  - 2.5.1 Valeur de retour d'une fonction . . . 12
  - 2.5.2 Passage de paramètres : par valeur ou par adresse ? . . . 12
- 2.6 Directives du préprocesseur et prototypage . . . 13
  - 2.6.1 Directive #include . . . 13
  - 2.6.2 Directive #define . . . 13
  - 2.6.3 Directive typedef . . . 14
  - 2.6.4 Prototypage . . . 14
- 2.7 Les structures de contrôle : Tests et Boucles . . . 14
  - 2.7.1 Expression booléenne . . . 14
  - 2.7.2 Tests imbriqués . . . 15
  - 2.7.3 Boucles . . . 16
- 2.8 Tableaux et pointeurs . . . 17
  - 2.8.1 Tableaux numériques . . . 17
  - 2.8.2 Tableaux et chaînes de caractères . . . 18
  - 2.8.3 Représentation d'un tableau en mémoire . . . 18
  - 2.8.4 Pointeurs et allocation dynamique . . . 18
- 2.9 Structures et Unions . . . 21
  - 2.9.1 Structures . . . 21
  - 2.9.2 Unions . . . 22
- 2.10 Entrées-sorties les plus utilisées . . . 23

2.10.1 printf . . . 23
2.10.2 scanf . . . 24
2.11 Accès aux fichiers . . . 24
2.11.1 Ouverture-création-fermeture . . . 25
2.11.2 Lecture-écriture . . . 25
2.11.3 Positionnement et fin de fichier . . . 26
2.12 Paramètres de la fonction main() . . . 26
2.13 Diagnostiquer les erreurs. . . . 26
2.13.1 L'inévitable Segmentation fault . . . 26
2.13.2 Un résultat de calcul inattendu . . . 27

3 Compléments de cours **29**
3.1 Le fichier Makefile . . . 29
3.2 Le debugger . . . 30
3.3 Les outils de debug spécifiques à l'allocation dynamique . . . 30
3.4 Interfaçage avec du code Fortran . . . 31

4 Annexes **33**
4.1 Fonctions mathématiques du C . . . 33
4.2 Fonctions de manipulation de chaînes de caractères . . . 34
4.3 Fonctions d'accès mémoire . . . 34
4.4 Options principales de gcc . . . 35

Chapitre 1

Bases de programmation

1.1 Langage de programmation

Un ordinateur est une machine « rudimentaire », ne sachant basiquement répondre qu'à très peu d'ordres :
- placer ou extraire une donnée d'une zone mémoire
- additionner, soustraire et multiplier en binaire uniquement
- comparer des nombres

Son atout est qu'il peut être **programmé**, c'est à dire qu'on peut lui établir, à l'avance, une séquence d'ordres (ou d'instructions) à exécuter. Or l'ordinateur est très rapide et peut répéter un grand nombre de fois une séquence sans se tromper. D'où sa fabuleuse **puissance**. Mais sans programmes (séquences d'instructions), un ordinateur n'est rien.

L'ordinateur ne comprenant que des ordres codés en binaire, des langages dits « évolués » ont été mis au point pour faciliter leur programmation. Ainsi sont nés dans les années 60, le FORTRAN (FORmula TRANslator) pour le calcul scientifique et le COBOL pour les applications de gestion. Puis, pour des besoins pédagogiques principalement, ont été crées le BASIC et le PASCAL au début des années 70. Ce dernier (comme le C) favorise une approche méthodique et disciplinée (on dit « structurée »). Le C a été développé conjointement au système d'exploitation Unix, dans les Laboratoires BELL, par Brian W. Kernighan et Dennis M. Ritchie, qui ont défini en 78, dans « The C Language », les règles de base de ce langage. Le but était de combiner une approche structurée (facilitant la programmation) avec des possibilités proches de celles du langage machine pour une efficacité maximale en exécution (quitte dans ce cas, à passer plus de temps de programmation). Ceci tout en restant standard, pour pouvoir être porté facilement sur n'importe quelle machine. Ce langage a été normalisé en 88 (norme ANSI).

1.2 Compilation et édition de lien

La plupart des langages informatiques sont des langages compilés, par opposition aux langages interprétés, comme le Basic par exemple, ou les langages de script, dont les instructions ne sont traduites qu'au moment de l'exécution.

1.2.1 Compilation

La compilation traduit un fichier **source**, créé à l'aide d'un éditeur de texte, en un fichier **objet**. Un fichier objet est un fichier contenant lui aussi des instructions et des données, mais en langage machine, sous une forme exploitable par l'éditeur de lien (cf. section 1.2.2), qui lui devra créer l'**exécutable** (ou binaire). Chaque donnée globale ou fonction définie dans un fichier objet possède un **nom symbolique**. Chaque référence à un symbole extérieur au fichier objet est appelée **référence externe** (Le compilateur ne peut pas résoudre ces références externes puisqu'il ne travaille que sur un seul fichier source à la fois). Pour

répertorier ces symboles et ces références extérieures, le fichier objet possède une table des symboles.

Pour connaître les symboles définis dans un fichier objet, utiliser la commande **nm** *suivie du nom du fichier objet. La première colonne donne l'adresse de chaque symbole ou référence, la dernière, le nom du symbole. La seconde colonne, elle, donne le type du symbole :*

- *T indique une définition globale*
- *U indique une référence externe*
- *d indique une définition de donnée locale*
- *b indique une définition locale de bss (aire de données non initialisée).*

Les **librairies** ne sont que des cas particuliers de fichiers objet, archivant des fonctions très souvent utilisées. Tout langage a les siennes, et tout programmeur peut s'en constituer de nouvelles. Par convention, les noms de librairies ont la forme suivante : lib*name.sfx*, où *name* est une chaîne de caractères qui identifie la librairie, et *sfx*, un suffixe caractéristique du type de la librairie : *sfx* vaut **.a** si la librairie est une archive (ou librairie statique, voir ci-dessous), **.sl** ou **.so** s'il s'agit d'une librairie partagée (ou dynamique). On se réfère à une librairie en l'appelant par son nom, dépouillé du suffixe. Par exemple, on se réfère à la librairie standard C par $libc$.

Différence entre librairie statique et librairie partagée :
Une librairie est dite **statique** si, lorsque des fonctions de cette librairie sont utilisées dans un programme, le code de ces fonctions est inclus dans l'exécutable du programme. La présence de la librairie n'est alors pas nécessaire pour l'exécution du programme puisuqe toute l'information nécessaire est disponible dans l'exécutable. Au contraire, dans le cas d'une librairie **partagée**, le code n'est pas répliqué dans l'exécutable, mais est chargé en mémoire au moment de l'exécution du programme. Les avantages sont, d'une part, que l'exécutable est plus petit, et d'autre part, et que le code correspondant n'est stocké en mémoire qu'une seule fois quelque soit le nombre d'utilisateurs qui l'utilise (Si un programme utilise des fonctions dont le code a déjà été porté en mémoire par un autre programme, il s'en servira directement). Les inconvénients sont la présence obligatoire de la bibliothèque pour l'exécution du programme et quelques règles plus strictes de mise au point que nous ne détaillerons pas ici.

Pour connaître la liste des fichiers objet archivés dans une bibliothèque, utiliser la commande **ar -t** *suivi du nom de la librairie. Pour archiver un objet dans une bibliothèque, utiliser la commande* **ar -r**. *Certaines versions de la commande ar requièrent que la commande ranlib soit exécutée postérieurement à l'archivage, afin de mettre à jour la table des symboles. La plupart des versions actuelles de ar le font automatiquement.*

À noter : Sous Unix, le compilateur C se nomme généralement **cc** *ou* **gcc** *(documentation en annexe).*

1.2.2 Édition de lien

L' éditeur de lien utilise les tables des symboles des différents fichiers objets et tente d'assortir les références externes aux définitions globales. C'est lui qui établit les liens entre les différents objets. Si l'éditeur de lien ne peut pas lier une référence externe (c'est à dire, s'il ne peut trouver la description du code lui correspondant), il affiche un message sur la sortie d'erreur standard.

Sous Unix, l'édition de lien est gérée par la commande **ld** *quelque soit le langage de programmation. Toutefois certains compilateurs possèdent une option pour invoquer directement l'édition de lien. Ceci peut amener une certaine confusion, car la même commande est alors utilisée pour la compilation et l'édition de liens, aux options d'exécution près.*

L'édition de lien est suivie de la commande **strip** lorsque le programme est considéré comme stable. Cette commande Unix fait un peu de nettoyage dans l'exécutable en enlevant les informations destinées au debugger (voir section 3.2). Ceci permet de réduire la taille de l'exécutable.

1.2.3 Avantage et inconvénient des langages compilés :

Avantage : L' exécution est beaucoup plus rapide, notamment lorsque le programme contient des boucles. En effet, dans le cas d'un langage interprété, les lignes de commande constituant le corps de la boucle vont être retraduites en langage machine autant de fois que la boucle va s'exécuter.

Inconvénient : Les fichiers objet et exécutable sont dépendant du type de machine sur lequel ils ont été générés (On ne peut exécuter sur une station SUN, par exemple, un programme compilé sur une IBM).

1.3 Programmation structurée

1.3.1 Modularité des programmes

Quelque soit le langage utilisé pour programmer, il est important de respecter certaines règles d'organisation, si on veut être efficace. En science, comme dans beaucoup de domaines, un certain nombre d'opérations de base sont communes à de nombreux calculs. Il est important de les identifier pour optimiser les développements. Le temps « perdu » pour généraliser un temps soit peu la programmation d'une fonction de base est largement rentabilisé lors de futurs développements. Elle constituera ultérieurement une brique de base, comme dans un jeu de construction, qu'il suffira d'assembler, voire d'enrober, pour créer rapidement des programmes plus complexes.

Règles de bases :

- isoler les fonctions fréquemment utilisées dans des modules à part et les rassembler par thèmes. On pensera notamment aux fonctions de lecture−écriture sur disques (appelées fonctions d'entrées−sorties), aux méthodes statistiques et numériques courantes, etc. (cf. bibliothèques)
- prendre le temps de tester (on dit aussi debugger en anglais) chaque module qui deviendra une des bases des programmations futures. Vous pourrez ainsi vous concentrer ensuite seulement sur le code nouveau, ce qui constitue un gain de temps appréciable lors du débogage, le champ d'investigation étant moins grand.
- prendre le temps de documenter un minimum chaque module, faute de quoi, il deviendra rapidement inexploitable, même par celui qui l'a écrit. Au minimum : en entête de fichier source, le nom du programmeur (on travaille de plus en plus rarement tout seul !) puis, pour chaque fonction, une description des paramètres passés ou retournés, et une brève description de ce qu'elle fait)
- essayer de structurer les programmes (la modularité va dans ce sens). Éviter les « goto » même s'ils existent dans de nombreux langages. Une boucle ou une alternative (si…alors…sinon…) doivent avoir un début et une fin clairement identifiés. S'il n'en est rien, vous vous exposez à bien des déboires, d'ici à quelques mois, lorsqu'il faudra amender le code (un programme est rarement définitif, il subit des mise à jour régulières).

1.3.2 Bibliothèques

Une librairie (ou bibliothèque) est un ensemble de fonctions (ou de modules) **testées** et archivées pour être **réutilisées**. Il est vivement conseillé de se constituer des bibliothèques à thème (graphiques, entrée−sorties, fonctions mathématiques, statistiques, etc.) documentées. Ce sera un atout certain pour des développements rapides et efficaces.

1.4 Documentation - Lisibilité des sources

Si cette étape vous paraît futile dans l'instant, il vous faudra peu de temps pour réaliser que vous ne lui avez pas accordé suffisamment d'importance. Certes, beaucoup de petits programmes s'écrivent dans l'urgence (personne n'échappe à la règle). Prévoyez toujours cependant un minimum de commentaires dans vos programmes, car il est beaucoup plus fréquent qu'on ne le croit que celui ci soit réutilisé comme base d'un nouveau développement rapide.

La programmation structurée est d'une grande aide, car un programme bien écrit limite (mais attention ne supprime jamais) le besoin de documentation. N'hésitez pas à donner des **noms explicites** à vos variables et à vos fonctions. Ce n'est pas une perte de temps car ça aide beaucoup pour la lisibilité du programme. Toute pièce de code (module de programme) trop longue doit impérativement être commentée si on ne veut pas passer des heures à retrouver ce qu'elle fait. Là encore, plus le programme est modulaire, plus il a de chance d'être lisible sans grand effort de documentation (on retrouve plus facilement à quoi servent 5 lignes consécutives de programmes que 50 !).

Chapitre 2

Langage C

Pourquoi apprendre le langage C ?

Le langage C est un langage structuré, avec toutes les possibilités des autres langages structurés. Mais il permet également (avec son extension C++) de gérer des objets. À l'inverse, il permet également une programmation proche du langage machine, ce qui est souvent nécessaire lors d'interfaçage entre l'ordinateur et son environnement extérieur (cas de l'instrumentation par exemple). Mais son principal avantage est que ces trois types de programmation peuvent être combinés dans un même programme, tout en restant portable (compilable et exécutable sur tout ordinateur). Le langage C a néanmoins un inconvénient, c'est d'être un peu plus complexe d'utilisation (mais uniquement du fait de ses nombreuses possibilités, notamment l'allocation dynamique).

Ce cours n'a pas la prétention de couvrir tout le C. Il existe de très bons livres sur le sujet. Il se veut plutôt une introduction à la programmation C en mettant autant que possible en avant les erreurs fréquentes. Il se veut aussi un support de cours et un fascicule de référence « léger » pour les débutants.

2.1 Règles de syntaxe du C

- Chaque instruction d'un programme en C se termine par le caractère « ; »
- Une instruction peut s'écrire sur plusieurs lignes
- Un bloc d'instructions est délimité par les caractères { et }
- Les compilateurs sont sensibles aux minuscules et aux majuscules (c'est à dire que deux variables Abc et ABC seront différentes pour le C)
- Toute chaîne de caractères comprises entre les symboles /* et */ est assimilée à un commentaire (attention, on ne peut imbriquer plusieurs commentaires)
- Un appel de fonction s'écrit : $fonction(par_1, par_2, par_3, ...)$ où par_1, par_2, par_3,..., par_i sont les paramètres de la fonction à appeler.

2.2 Les types

Les différents types de variables connus du C sont :

Type	Taille mémoire usuelle	Description	Intervalle
char	1 octet	entier	[-128,127]
short int	2 octets	entier	[-32 768, 32 767]
long int	4 octets	entier	+/- 2 milliards
float	4 octets	réel	$[3.4 \times 10^{-38}, 3.4 \times 10^{+38}]$
double	8 octets	réel	$[1.7 \times 10^{-308}, 1.7 \times 10^{+}308]$
long double	16 octets	réel	$[3.4 \times 10^{-4932}, 3.4 \times 10^{+}4932]$

On utilise très souvent le type *int* qui est une abbreviation, selon la machine, des types *short int* ou *long int* (dans la majorité des cas, un *int* est équivalent à un *long int*). Les types $char$ et les différentes déclinaisons du type entier peuvent être précédés du mot clé **unsigned** qui spécifie qu'on travaille sur une donnée non signée. Dans le cas d'une variable de type char, par exemple, l'intervalle des valeurs qu'il est possible de stocker devient alors [0,255].
La déclaration d'une variable se fait en donnant son *type* suivi de son *nom*. Par exemple :

```
int nb_pts;
char caractere;
double resultat;
```

Si plusieurs variables du même type doivent être déclarées, on simplifie l'écriture en listant leurs noms, séparés par une virgule, derrière la déclaration de type :

```
int nb_pts, i, j;
```

Une variable peut être **locale** ou **globale** selon qu'elle est déclarée à l'intérieur d'une fonction ou non. Si elle est locale, elle n'est visible qu'á l'intérieur de la fonction qui la définie. Si elle est globale, elle est toujours visible, quelle que soit la partie du programme dans laquelle on se trouve. On peut donc trouver dans un programme plusieurs variables locales portant le même nom, mais n'ayant aucun lien entre elles. A l'inverse, toutes les instance d'un même nom de variable peuvent correspondre à une seule et même donnée (variable globale).

2.3 Les opérations simples

2.3.1 L'affectation

L'affection est tout simplement le fait de donner une valeur à une variable. On écrit : $variable = valeur$;
Toutefois, il est plus juste de lire le signe « = » comme « reçoit ». En effet, on peut écrire

```
a = a + 2;
```

c'est à dire « a reçoit l'ancienne valeur de a à laquelle on a ajouté 2 ». C'est toujours la partie gauche de l'expression qui reçoit la partie droite.

2.3.2 Les opérateurs arithmétiques

Comme tous les langages, le C contient des opérateurs qui permettent d'effectuer des opérations arithmétiques sur les variables. Les cinq opérateurs du C sont :
- l'addition +
- la soustraction -
- la multiplication *
- la division /
- le modulo %

Il n'existe pas d'opérateur de puissance (c'est une fonction mathématique !).

Attention ! La division, suivant les types des opérandes, donne un résultat entier ou flottant. Ainsi :
- 7/2 retourne 3 (division euclidienne)
- 7.0/2 ou 7/2.0 ou 7.0/2.0 retournent 3.5

2.3.3 Les raccourcis

Il existe pour ces opérateurs des raccourcis qui permettent d'écrire un code plus concis, en principe plus optimisé et souvent moins lisible (à manier donc avec précaution). Ainsi, si une variable apparaît des deux côté du signe $=$, on peut écrire :

– $i \mathrel{+}= n$, équivalent de $i = i + n$
– $i \mathrel{-}= n$, équivalent de $i = i - n$
– $i \mathrel{*}= n$, équivalent de $i = i * n$
– $i \mathrel{/}= n$, équivalent de $i = i/n$
Dans le cas d' incréments de valeur 1 ou -1, on peut aussi écrire :
– $i{+}{+}$ ou $+{+}$i, équivalent à $i = i + 1$
– $i{-}{-}$ ou $-{-}$i, équivalent à $i = i - 1$

2.3.4 Les opérateurs de traitement de bits

Pour les futurs instrumentalistes ! Ils ne s'appliquent qu'aux entiers (signés ou non) et au type $char$.
– & ET bit à bit
– | OU inclusif bit à bit
– ˆ OU exclusif bit à bit
– $<<$ décalage à gauche
– $>>$ décalage à droite
– ˜complément à un

2.3.5 Les opérateurs de comparaison

Il est possible de comparer deux variables grâce aux opérateurs suivants :
– $<$ pour strictement inférieur
– $>$ pour strictement supérieur
– $<=$ pour inférieur ou égal
– $>=$ pour supérieur ou égal
– != pour différent
– == pour égal
Attention ! Une source fréquente d'erreur vient de la confusion entre ce dernier opérateur et l'opérateur d'affectation !

2.3.6 L' opérateur & - Notion d'adresse

L'opérateur &, permet à tout moment de connaître l'emplacement en mémoire d'une variable (c'est à dire son adresse mémoire). Lorsque vous déclarez une variable

```
long int i;
```

le compilateur réserve un espace en mémoire de 4 octets, qui, contiendront la valeur de i (ou encore, contenu de i). Mais ces quatre octets sont référencés par une adresse mémoire pour savoir à tout moment où les retrouver. C'est cette information que retourne l'opérateur &.

L'opérateur & est valable pour tous les types de variables. I l se place AVANT la variable dont on veut connaître l'adresse. Par exemple $\&a$ désigne l'adresse de la variable a.

2.4 Les conversion de type

Il existe des opérateurs unaire, appelé **cast**, permettant de spécifier explicitement une conversion de type. Il suffit pour cela de faire précéder le nom de la variable à convertir par le nouveau type, encadré par des parenthèses. Par exemple :

```
int a;
float f;

a = (int) f;
```

convertit le nombre flottant f en nombre entier, avant de l'affecter à a. Il faut être conscient qu'il ne s'agit pas d'une simple affectation, la représentation mémoire d'un entier étant complètement différente de celle d'un flottant.

2.5 Programme principal et appel de fonctions

La syntaxe d'appel d'une fonction est la suivante :
$fonction(par_1, par_2, par_3, ...)$;
où par_1, par_2, par_3, par_i sont les paramètres de la fonction à appeler.

Le C donne au programme principal l'allure d'une fonction particulière appelée **main()**. Ses arguments, s'ils sont specifiés décrivent la ligne de commande (nous les détaillerons plus loin). Un exemple de programme très simple en C, est :

```
main()
{
    printf("Bonjour");
}
```

La fonction $printf()$ (fonction standard du C) affiche à l'écran une expression. Ici, le texte affiché est simplement «Bonjour».

2.5.1 Valeur de retour d'une fonction

Une fonction peut retourner une valeur. Ainsi on peut écrire :

```
  c = somme(a, b);
```

où la fonction $somme()$ calcule la somme de deux valeurs et retourne le resultat pour être placé dans la variable c. La fonction se terminera, dans ce cas, par l'instruction **return** suivi de la valeur qui doit être retournée :

```
float somme(float a, float b)
{
    return a+b;
}
```

Il existe un type particulier aux fonctions, c'est le type **void**. Il signifie que la fonction ne retourne aucune valeur (par défaut, une fonction retourne un entier).

2.5.2 Passage de paramètres : par valeur ou par adresse ?

En C, les paramètres peuvent être passés à la fonction soit par valeur (c'est à dire qu'on passe le contenu de la variable), soit par adresse (on passe l'adresse mémoire où est stockée la variable). Dans d'autres langages, le passage se fait systématiquement par adresse, mais cet aspect des choses est masqué. Soit x une variable. Quand utiliser x ou $\&x$ lors d'un passage de paramètre ?
La règle est simple :
Si la valeur de la variable n'est pas modifiée dans la fonction, elle peut être passée par valeur. Sinon, elle **doit** *être passée par adresse !*

Les tableaux (et donc les chaînes de caractères) sont **toujours** passées par adresse, puisque donner le nom du tableau comme argument revient à donner son adresse (adresse de la première valeur du tableau).

Exemples :
– passage par valeur :

```
#include <math.h>

double sinus(double x) {
    return(sin(x));
}

main()
{
    double pi, resultat;

    pi = 3.14159;
    resultat = sinus(pi);
}
```

– passage par adresse :

```
void set(double *x, double val) {
    *x = val;
}

main()
{
    double pi;

    set(&pi, 3.14159);
}
```

2.6 Directives du préprocesseur et prototypage

2.6.1 Directive #include

Vous noterez qu'on a rajouté, au début du programme précédent, la ligne « #include. . .». Cette directive demande au compilateur (en fait au préprocesseur[1]) d'inclure le fichier *math.h*. C'est un fichier livré avec tous les compilateurs C qui décrit de nombreuses fonctions mathématiques. (Vous trouverez facilement une liste des fonctions et des fichiers à inclure dans les fichiers d'aide de vos compilateurs ou dans les manuels). L'intérêt de ce type de fichier est d'une part d'inclure de nouvelles définitions dans votre programme, et d'autre part de décrire les arguments des fonctions externes au programme (**prototypage**), et donc d'aider le compilateur à vérifier plus à fond votre code. Ces fichiers rassemblent généralement des définitions de fonctions liées autour d'un thème (entrées-sorties, fonctions mathématiques, etc.).

2.6.2 Directive #define

Cette directive sert à définir une équivalence. Ainsi, si je trouve, dans une entête de programme, la ligne :

```
#define PI 3.14159
```

toute occurence du texte PI dans la suite du programme sera automatiquement remplacé par la valeur 3.14159 au moment de la compilation. L'intérêt est de pouvoir modifier aisément toutes les occurences de PI dans vos programmes le jour où vous souhaitez, par exemple, accroître la précision de vos calculs en spécifiant plus de décimales. Un autre intérêt est d'améliorer la lisibilité d'un programme sans passer forcément par des variables intermédiaires.

[1] Première phase de la compilation qui consiste à substituer aux macros le code équivalent

2.6.3 Directive typedef

Elle permet de définir un nouveau type. Ce peut être une simple équivalence avec un type existant ou un type plus complexe comme des enregistrements structurés que nous verrons plus loin dans le cours. Exemple :

```
typedef char BYTE
```

défini le type BYTE, qui peut améliorer la lisibilité d'un programme sensé manipuler des octets. Les déclarations de variables auront alors la forme suivante :

```
BYTE b1;
```

Le nouveau type devient un type à part entière est suit donc les mêmes règles syntaxiques que les types prédéfinis.

2.6.4 Prototypage

Il s'agit en fait d'une simple description du type de données passées à une fonction et retournées par celle-ci. L'intérêt est de faciliter la mise au point des programmes, car le compilateur testera la compatibilité des types passés à la fonction avec ceux déclarés lors du prototypage. De plus il évitera au compilateur de donner automatiquement le type entier (type par défaut) au données dont il de connait pas la nature.
Exemple de prototypage : soit la fonction précédemment définie :

```
void set(double *x, double val) {
    *x = val;
}
```

Prototyper set revient à insérer, en entête de fichier (en fait en préalable à toute utilisation de la fonction), la ligne :

```
void set(double *, double);
```

Ce qui aurait d'ailleurs dû être fait dans l'exemple précedent pour que le compilateur sache que la constante 3.14159 est en fait un double. On aurait pu aussi spécifier « (double)3.14159 » lors du passage de paramètre.

2.7 Les structures de contrôle : Tests et Boucles

L'expression d'une condition se fait de la façon suivante (Attention, les parenthèses sont obligatoires !) :

```
if (expression booleenne)
    instruction;
else
    instruction;
```

Si l'évaluation de l'expression donne une valeur nulle, alors l'expression est fausse, sinon elle est vraie.

2.7.1 Expression booléenne

Une expression booléenne peut être constituée d'

- une $variable$. Si sa valeur est nulle, le résultat du test est faux, sinon il est vrai. Pour nier une variable, utiliser l'opérateur « !» :
  ```
  if (!recu) printf("rien recu\n");
  ```
  Le message est affiché si $recu$ contient 0.
- un $test$, soit une expression constituée à l'aide d'opérateurs de comparaison. Par exemple :
  ```
  if (i<10) i++;
  ```
  La variable i sera incrémentée seulement si i est inférieur à 10.

– une *suite de variables et de tests* combinés à l'aide des opérateurs logiques $\&\&$ (ET logique) et $||$ (OU logique). Par exemple :
`((a>3) && (a<=5))`
se lit « a supérieur à 3 et a inférieur ou égal à 5 », soit « a appartient à l'intervalle $]3, 5]$. »
`((a<3) || (a>=5))`
« a appartient au domaine $]-\infty, 3[\cup[5, \infty[$. »

Si le traitement à effectuer nécessite plusieurs instructions, on utilise des accolades pour donner les limites du bloc d'instruction. Par exemple :

```
if (!recu) {
    ....
    ....
}
else {
    ....
    ....
}
```

2.7.2 Tests imbriqués

Si on doit imbriquer plusieurs instructions *if ... then ... else*, il est souvent préférable d'utiliser la structure **switch** :

```
   switch (expression)
   {
      case cste1: instructions
            break;

      case cste2: instructions
            break;
      ....

      case csteN: instructions
            break;

      default: instructions
            break;
   }
```

où la liste des *csteXXX* représente les valeurs que peut prendre le résultat de l'expression. Suivant la valeur prise par la variable (ou l'expression) testée, l'une des instances *case* sera exécutée, et ce jusqu'à rencontrer un *break*. Si celui ci n'est pas présent à la fin du bloc d'instruction correspondant au $case$, les instructions appartenant au *case* suivant seront exécutées également. Par exemple :

```
/* fonction verifiant si l'argument passe est une voyelle */

int voyelle(char c)
 {
  int test;

  switch(c)
   {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
```

```
    case 'u':
    case 'y': test = 1; /* vrai */
                break;
    default : test = 0; /* faux */
   }
   return test ;
}
```

Si ce peut être très utile, attention, c'est aussi une source d'erreur fréquente !

Le mot clé *default* signifie « dans tous les autres cas ». C'est le mode par défaut pour toutes les éventualitées non listées. Attention, l'expression évaluée ne peut être que de type entier ou caractère.

2.7.3 Boucles

Il existe deux types de boucles en C : la boucle **for**, qui est beaucoup plus riche en possibilités que dans les autres langages, et la boucle *while*. L'une et l'autre peuvent le plus souvent être utilisé indifféremment.

Boucle FOR

La syntaxe est la suivante :

```
for (initialisation; condition de continuation; increment)
    instruction;
```

Par exemple :

```
for (i=0; i < 10; i++)
{
        .....
}
```

exécute 10 fois (i variant de 0 à 9) les instructions qui suivent. La partie initialisation et la partie incrémentation, peuvent contenir plusieurs instructions séparées par des virgules. Soit :

```
for (i=0, j=-5; i < 10; i++, j+=2)
```

Elles peuvent aussi être vides (pas forcément les deux en même temps !).

```
for (; i < 10;)
```

L'intérêt de cette présentation en ligne (initialisation + condition + incrément) est indéniable au moment de la programmation. Elle répertorie les étapes clés et évite ainsi des oublis (plus difficile d'écrire des boucles sans fin).

Boucle WHILE

Elle est utilisée plus particulièrement dans les cas de boucles qui doivent s'exécuter au moins une fois, c'est à dire où le test de sortie s'effectue en fin. C'est le cas dans l'écriture suivante :

```
do {
    ....
}
while (condition);
```

qui peut se lire : exécuter les instructions de la boucle tant que la condition de continuation est vraie. Une boucle *while* peut aussi s'écrire :

```
while (condition)
{
    ....
}
```

La boucle est exécutée tant que la condition est vraie. La séquence d'instruction n'est pas forcément exécutée, si la condition est fausse d'emblée.
Deux instructions qui peuvent être utiles également dans la programmation des boucles : les instructions break et continue.

Instruction break

Elle provoque la sortie immédiate de la boucle en cours. Attention, elle est limité a un seul niveau d'imbrication.

```
for (i=0;i<10;i++)
{
    ....;
    if (erreur) break;
}
```

Instruction continue

Cette instruction provoque le passage à la prochaine itération d'une boucle. En cas de boucles imbriquées, elle permet uniquement de continuer la boucle la plus interne.

```
for (i=0;i<10;i++)
{
    if (i==j) continue;
    ......;
}
```

équivaut à :

```
for (i=0;i<10;i++)
    if (i!=j) {
         ......;
    }
```

2.8 Tableaux et pointeurs

2.8.1 Tableaux numériques

Un tableau permet de regrouper plusieurs éléments d'un même type. Un tableau se déclare en rajoutant la dimension à la suite de la déclaration de variable. Ainsi

```
int tab[100];
```

crée un tableau de 100 entiers. Attention, **la numérotation commence à zéro !** Le premier élément du tableau est donc $tab[0]$, et le dernier $tab[99]$. Un tableau à deux entrées se déclarera de la façon suivante :

```
int tab[100][100];
```

Le premier élément du tableau est $tab[0][0]$, le dernier $tab[99][99]$.
La même logique est utilisée quelque soit le nombre de dimensions.

2.8.2 Tableaux et chaînes de caractères

Une chaîne de caractère est, en C, un cas particulier du tableau de caractères. On déclare donc une chaîne de la même façon qu'un tableau. Seule exigence : la fin de la chaîne de caractères, est spécifiée, en mémoire, par le caractère '\0' (code ASCII 0). Ce caractère doit être compté lorsqu'on défini la taille de la chaîne.

```
char text[10] = "123456789";
```

Il y a 9 caractères dans la chaîne, plus le caractère de terminaison '\0'. On réserve donc 10 caractères en mémoire. Si le caractère de terminaison de chaîne n'est pas présent, on parle de tableau de caractère, et on ne peut lui appliquer aucune des fonctions spécialisées dans le traitement des chaînes de caractères. Par exemple :

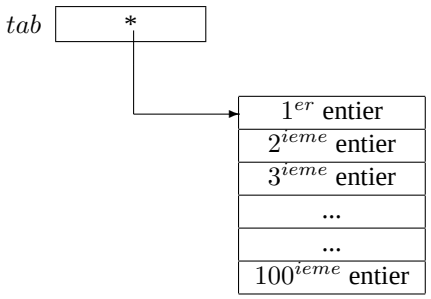
```
char text[3];

text[0] = 'A';
text[1] = 'B';
text[2] = 'C';
```

est une description correcte mais ne défini pas une chaîne. Il s'agit seulement d'un tableau de caractères auquel on accèdera élément par élément, comme un tableau numérique, jamais dans son ensemble.

2.8.3 Représentation d'un tableau en mémoire

La représentation mémoire d'un tableau en C, est la suivante :

Soit tab un tableau d'entiers, de taille 100. Le compilateur va d'abord réserver une zone mémoire qui contiendra l'adresse de début du tableau (soit l'adresse de la première valeur du tableau). Ensuite il réservera une zone de taille équivalente à 100 fois la taille d'un entier.



Cette séquence peut être reconstruite instruction par instruction. Dans quel intérêt ? Pour réserver soi même la place en mémoire lorsque la taille du tableau n'est pas connue à l'avance, la solution de surdimensionner atteignant rapidement ses limites. C'est ce qu'on appelle l'allocation dynamique.

2.8.4 Pointeurs et allocation dynamique

Dans le paragraphe précédent, la zone allouée par le compilateur pour stocker l'adresse du premier élément d'un tableau est appelé un **pointeur**. Un pointeur ne contient pas une valeur exploitable directement ; il sert à stocker une adresse mémoire. On le défini en faisant précéder le nom par une étoile. Exemple de déclaration d'un pointeur sur un tableau d'entiers :

```
int *tab;
```

Mais déclarer un pointeur ne veut pas dire qu'on a réservé de la place pour stocker des données. On a seulement réservé l'emplacement pour stocker l'adresse du premier élément du tableau ! Pour réserver l'emplacement des données, on dispose en C de deux fonctions :

– **malloc()** (pour *m*emory *alloc*ation) réserve une place mémoire de la taille (en octets) du paramètre qui lui est passé, et retourne l'adresse de début de la zone allouée. Par exemple :

```
#include <malloc.h>
char *txt;

txt = malloc(100);
```

malloc() réserve 100 octets en mémoire et renvoie l'adresse du premier. Pour une chaîne de caractère, il est facile de calculer la taille de la mémoire nécessaire, vu que l'unité de stockage est l'octet. Pour tous les autres types, il faut connaître la taille en mémoire des différents types de données. Pour simplifier, le C dispose d'une fonction **sizeof()**, qui renvoie la taille en octet correspondant au type demandé. Par exemple, $sizeof(int)$ renvoie usuellement 4 (mais cette taille dépend de la machine sur laquelle vous travaillez). On écrit dont généralement :

```
int *tab;

tab = (int *)malloc(100*sizeof(int));
```

ce qui permet en outre de s'affranchir de tous les problèmes de portabilité du code.

– **calloc()** est la deuxième fonction permettant d'allouer de la place mémoire. Sa syntaxe diffère légèrement de celle de $malloc()$. Son intérêt est d'initialiser automatiquement à zéro tous les octets alloués. Ainsi :

```
tab = (int *)calloc(100,sizeof(int));
```

est équivalent à :

```
tab = (int *)malloc(100*sizeof(int));
for (i=0; i<100; i++) tab[i]=0;
```

Revenons à présent sur la notion de pointeur. Le type pointeur est un type à part entière. On peut donc faire des opérations ou des comparaisons, sachant que :

– un incrément sur un pointeur admet pour unité la taille de son élément de base. Par exemple, dans le cas suivant :

```
int *tab1, *tab2;

tab2 = tab1 + 2;
```

si $tab1$ pointait sur le premier élément d'un tableau d'entier, alors $tab2$ pointera sur le troisième. Ce qui facilite grandement les opérations car il serait fastidieux d'avoir à calculer un décalage en terme d'octets, la taille du déplacement variant selon le type de donnée

– un pointeur peut lui même référencer un autre pointeur (adressage indirect).

– l'égalité de deux pointeurs implique qu'ils adressent la même zone mémoire. Ce qui signifie qu'on peut utiliser l'un ou l'autre des pointeurs pour référencer les données, et que corollairement, les données modifiées par référence à l'un le sont aussi pour l'autre. Ça paraît évident, mais, dans un programme, on a vite tendance à oublier qu'on parle du même tableau dès qu'on y accède via deux pointeurs aux noms différents

– on peut avoir des tableaux de pointeurs. On déclare alors soit :

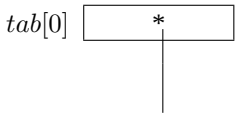
```
int *tab[7];
```

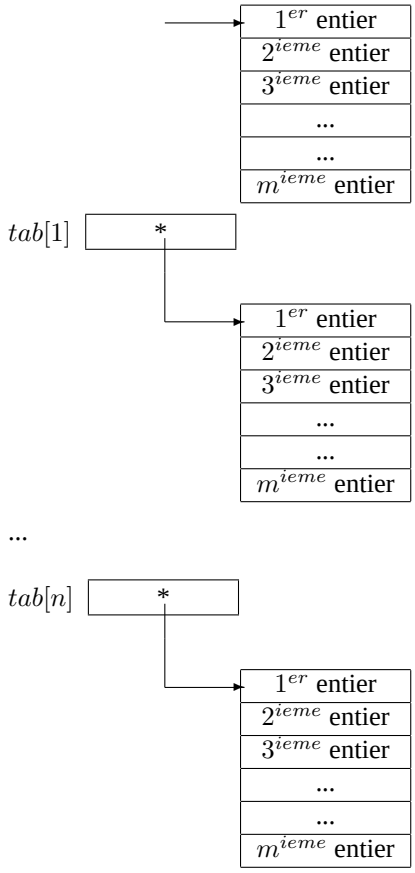
pour un tableau de 7 pointeurs, soit

```
int **tab;
```

si la taille du tableau n'est pas connu d'avance

Cette dernière possibilité est utilisée pour allouer dynamiquement des tableaux à double entrée. Un tableau à deux dimensions $tab[n][m]$ peut en effet être vu comme un tableau de n pointeurs, chacun des n pointeurs pointant sur un tableau de m valeurs, représentant chacun une ligne du tableau. Soit :





Ce qui s'écrit :

```
    /* rappel : n = nombre de lignes, m = nombre de colonnes */

int **tab;
int i;

        ....

tab = (int **)malloc(n*sizeof(int *));
for (i=0;i<n;i++)
    tab[i] = (int *)malloc(m*sizeof(int));
```

Si vous préférez que toute la mémoire allouée pour le tableau soit consécutive en mémoire (indispensable lors de l'interfaçage avec d'autres langages, comme par exemple le Fortran), rien ne vous empêche de le programmer ainsi (mais attention, plus le produit $m \times n$ est grand, plus le système aura de mal à trouver une telle zone mémoire à allouer).

Libération de la mémoire

Rien ne sert d'optimiser la gestion de la mémoire en utilisant l'allocation dynamique, si on ne peut libérer également toute place qui n'est plus requise. Il existe une fonction C pour ce faire. C'est la fonction **free()**, qui se syntaxe :

```
free(pointeur);
```

quelque soit le type de pointeur utilisé. Dans le cas des pointeurs, le type $void$ à une signification particulière. Recevoir dans une fonction un argument de type void * signifie qu'on ne connait pas à priori le type des données passées.

2.9 Structures et Unions

2.9.1 Structures

Une structure représente une collection de données, qui peuvent être de types différents, liées entre elles pour être manipulées facilement. C'est l'équivalent de l'enregistrement dans plusieurs autres langages. Par exemple :

```
struct individu {
    char *nom;
    char *prenom;
    int age;
    char sexe;
    char *adresse;
}
```

Une structure est déclarée par le mot-clé $struct$ suivi d'un nom (ici $individu$) qui identifie ce type de structure. On désigne l'un des éléments d'une structure par la forme $nomdestructure.element$. Par exemple :

```
struct Point {
    float x;
    float y;
}

main() {

    struct Point A;

    A.x = 0.0;
    A.y = 1.0;
}
```

La structure n'est pas un vrai type au sens du C. Les différentes opérations d'affectation, de comparaison, etc. s'appliquent à chacun de ses membres, pas à la totalité de la structure (sauf $sizeof()$). Par contre, on peut passer une structure comme argument à une fonction. C'est d'ailleurs un de ses grands atouts. Cela permet de grouper des paramètres entre eux, lorsqu'ils seront utiles à plusieurs fonctions.

On peut aussi initialiser cette structure à la déclaration, en la faisant suivre de la liste des valeurs des différents éléments :

```
    struct Point A = {0.0, 1.0};
```

Si la structure est couramment utilisée, elle peut faire l'objet d'une déclaration de type, à l'aide d'un $typedef$. Reprenons l'exemple précédent. Il s'écrit à présent :

```
typedef struct {
    float x;
    float y;
} Point;

main() {

    Point A;
```

```
    A.x = 0.0;
    A.y = 1.0;
}
```

Comme il y a des pointeurs sur des variables, il y a des pointeurs sur les structures. Ici aussi on se contente de faire précéder le nom d'une étoile, dans la déclaration. Mais il convient de noter la différence d'écriture pour accéder à un élément (ou champ) de la structure : le délimiteur « . » est remplacé par le symbole « -> ». Exemple :

```
struct Point {
    float x;
    float y;
}

main() {

    struct Point A;
    struct Point *pA;

    pA = &A;
    pA->x = 0.0;
    pA->y = 1.0;
}
```

Et de même, il y a des tableaux de structures. Soit :

```
struct Point {
    float x;
    float y;
}

main() {

    struct Point X[100];

    X[0].x = 0.0;
    X[0].y = 1.0;

    X[1].x = 1.0;
    X[1].y = 0.0;

    ....
}
```

2.9.2 Unions

La notion d'union, elle, définit un ensemble de données de type différents mais qui recouvrent toutes le même emplacement mémoire. Pour comprendre l'utilité de l'union, prenons un exemple : imaginons un programme qui peut admettre deux types de paramètres (chaîne de caractère ou entier). Le programme doit afficher l'argument :

```
union argument {
    char *chaine;
    int entier;
}
```

```
main() {

    char type;          /* type de l'argument : 'c' pour char et 'i' pour integer */
    union argument arg; /* contenu de l'argument */

    ....

    switch(type)
    {
        case 'c' :
            printf("chaine de caract\'ere : %s\n",arg.chaine);
            break;
        case 'i' :
            printf("entier : %d\n",arg.entier);
            break;
        default :
            printf("type d'argument incorrect\n");
            break;
    }
}
```

La syntaxe et les règles d'utilisation sont identiques aux structures.

N.B. *Pour pouvoir copier ou comparer des structures ou des tableaux aisément, on peut utiliser les fonctions d'accès mémoire données en annexe.*

2.10 Entrées-sorties les plus utilisées

Nous nous contenterons ici des deux fonctions principales d'entrée-sortie, à savoir la fonction permettant l'affichage de données à l'écran, et celle permettant de saisir des valeurs au clavier.

2.10.1 printf

La particularité de cette fonction est d'admettre un nombre d'argument variable. Le premier argument est une chaîne de caractère, appelée **format**, qui défini la mise en forme des données. Suivent ensuite les diverses variables qui doivent être affichées. Elle retourne le nombre de caractères imprimés.

```
printf(format_string, var1, var2, ..., varn);
```

La chaîne de caractère qui définit le format contient deux types d'objets :
- des caractères ordinaires qui seront recopiés tel quels à l'écran
- des spécifications d'affichage à l'emplacement choisi pour les différentes valeurs des arguments

Les spécifications d'affichage commencent toujours par le caractère % et se termine par un caractère de conversion qui spécifie le format d'affichage (affichage d'un entier, d'un flottant, etc.).

Ainsi on trouvera les différents formats suivant :

Format	Type d'impression	Type du paramètre en entrée
%d, %i	Affichage d'un nombre décimal	int
%u	Affichage d'un nombre décimal non signé	int
%f	Affichage d'un réel ($[-]m.ddddd$)	float ou double
%e, %E	Affichage d'un réel ($[-]m.dddExx$)	float ou double
%g, %G	Affichage d'un réel (équivalent à %f si l'exposant est < 4, à %e, sinon)	float ou double
%o	Affichage d'un nombre octal	int
%x, %X	Affichage d'un nombre hexadécimal	int
%c	Affichage d'un caractère isolé	int ou char
%s	Affichage d'une chaîne de caractères	char *
%p	Affichage d'un pointeur	void *
%%	Affichage du caractère %	

On peut placer, entre le caractère % et le format de conversion, les spécifications suivantes :
- le signe moins qui spécifie que l'affichage doit être cadré à gauche
- un nombre qui précise la largeur minimum du champ d'impression. Si ce nombre est réel, alors la partie décimale donne la *précision*. La précision indique, soit le nombre maximum de caractères à imprimer (cas d'une chaîne de caractères), soit le nombre de chiffres à imprimer pour les décimales (cas d'un réel), soit le nombre minimum de chiffres à imprimer (cas d'un entier)
- la lettre h si l'entier doit être converti en short ou la lettre l pour un long

En outre, le caractère \ permet de spécifier les caractères spéciaux (tels que les caractères d'alignement), appelés aussi caractères d'échappement :
- \n fin de ligne (passage à la ligne)
- \t tabulation horizontale
- \v tabulation verticale
- \b backspace
- \r retour en début de ligne
- \f saut de page
- \g alarme sonore
- \\ affiche le caractère \
- \ ? affiche le caractère ?
- \' affiche le caractère '
- \" affiche le caractère "

2.10.2 scanf

Cette fonction est assez similaire à la fonction printf() dans sa syntaxe. Mais cette fois on spécifie dans la chaîne de format, nom plus le format d'affichage mais le format de saisie des données. Les arguments suivants sont les noms des variables qui recevront les saisies.

```
scanf(format_string, &var1, &var2, ..., &varn);
```

Attention : les variables ici doivent être passées par adresse !

2.11 Accès aux fichiers

Nous nous limiterons aux fonctions d'accès de haut niveau, ou entrée-sorties bufférisées. À tout fichier est associé une structure de type **FILE**, définie dans <stdio.h> (pour STanDard Input-Output). Cette structure est initialisée lors de l'ouverture ou de la création d'un fichier et doit être fournie comme argument à toute fonction d'accès au fichier. Attention toutes ces fonctions retournent des codes d'erreur pour savoir si tout s'est bien passé. Encore faut-il les tester !

2.11.1 Ouverture-création-fermeture

La création et l'ouverture d'un fichier existant se font via la même fonction **fopen()**. Elle admet comme argument le nom du fichier à ouvrir (ou à créer) et le mode d'ouverture. C'est lui qui fera la différence entre les deux opérations. Le mode d'ouverture est une chaîne de caractère qui suit la logique :

- a : ouverture en mode append (ajout en fin de fichier). Le fichier est créé s'il n'existe pas
- r : ouverture en mode lecture seule
- w : ouverture en mode écriture seule. Si le fichier existe déjà, il est vidé de son contenu ; s'il n'existe pas il est créé
- r+ : ouverture en mode lecture-écriture
- w+ : ouverture en mode lecture-écriture. Si le fichier existe déjà, il est vidé de son contenu ; s'il n'existe pas il est créé
- a+ : ouverture en mode append (ajout en fin de fichier). Le fichier est créé s'il n'existe pas

Sur certains systèmes, il est possible d'ouvrir ou de créer un fichier en binaire à partir des fonctions décrites dans ce chapitre. Pour cela, il suffit de rajouter l'option b dans le mode d'ouverture, après les lettres a, r ou w et avant le signe + si celui-ci est utilisé.

La fonction $fopen()$ renvoit un pointeur sur une structure de type FILE qui sera passé aux autres fonctions d'accès au fichier.

Pour fermer un fichier, utiliser la fonction **fclose()**. Son seul argument est la structure de type FILE.

Exemple :

```
#include<stdio.h>
FILE *myfile;

myfile = fopen(nom_de_fichier, "r");
fclose(myfile);
```

2.11.2 Lecture-écriture

La fonction **fread()** lit un nombre n d'éléments dont on donne la taille unitaire en octets. La fonction retourne le nombre d'éléments lus. Inversement, la fonction **fwrite()** écrit un nombre n d'éléments dont on donne la taille unitaire en octets.
Déclarations respectives :

```
int fread(void *data, int size, int nb, FILE *id);
int fwrite(void *data, int size, int nb, FILE *id);
```

Ces fonctions sont utiles lorsqu'on cherche à lire ou écrire des données de même type (un tableau de nombre par exemple), mais ne conviennent pas tellement si vous souhaitez lire un fichier de configuration où l'utilisateur aura rentré une ligne du style « EDITOR = vi ». Pour ce faire, on utilisera plutôt les entrée-sorties formatées. Elles obéissent aux mêmes règles que leurs consoeurs vues précédemment $printf()$ et $scanf()$. Ce sont les fonctions **fscanf()** et **fprintf()**.

```
int fscanf(FILE *id, char *format, var1, var2, ...., varn);
int fprintf(FILE *id, char *format, var1, var2, ...., varn);
```

Il existe deux autres fonctions de lecture-écriture pour des chaînes de caractère, qui sont **fgets()** et **fputs()**.

```
int fgets(char *s, int max, FILE *id);
int fputs(char *s, FILE *id);
```

où s représente la chaîne à lire ou écrire, et max le nombre maximum de caractères à lire. La fonction $fputs()$ n'ajoute pas de caractère de fin de ligne '\n'. Il faut le spécifier si vous voulez un retour à la ligne.

2.11.3 Positionnement et fin de fichier

Pour se déplacer dans un fichier, on peut utiliser la fonction **fseek()**, qui opère soit en absolu à partir du début ou de la fin de fichier, soit en relatif, à partir de la position courante.
Déclaration :

```
int fseek(FILE *id, long offset, int reference_offset);
```

où $offset$ est la valeur du déplacement à effectuer (en positif ou négatif) à compter de la réference fournie en troisième argument. Cette référence vaut 0, si on part du début du fichier, 1, de la position courante et 2 de la fin de fichier.

Pour vérifier si on se trouve en fin de fichier, on utilise la fonction **feof()**, qui renvoie 1 si on est en fin de fichier, 0 sinon.

```
int feof(FILE *id);
```

2.12 Paramètres de la fonction main()

La fonction $main()$ peut admettre deux paramètres décrivant les arguments de la ligne de commande.

```
int main(int argc, char **argv);
```

où
- $argc$ est le nombre d'arguments présents sur la ligne de commande (c'est à dire qu'il inclus le nom du programme !)
- $argv$ est un tableau à double entrée donnant la liste des arguments

$argc$ vaut toujours au minimum 1, et $argv[0]$ contient le nom de l'exécutable. $argv[1]$ à $argv[n]$ donnent la liste des arguments spécifiés par l'utilisateur.

2.13 Diagnostiquer les erreurs...

Je n'ai pas la prétention de lister ici toutes les erreurs possibles, mais plutôt d'établir une liste des erreurs relativement fréquentes lorsqu'on débute la programmation en C (certaines même bien après !), et qui sont un brin difficiles à diagnostiquer. Quelques éléments d'enquête...

2.13.1 L'inévitable Segmentation fault

Le symptôme habituel est un programme qui affiche un « Segmentation fault » ou « Bus core dump » sans autre forme d'explication. Le debugger peut-être de quelque utilité, mais on peut déjà vérifier un certain nombre de points. Car nombreux sont les compilateurs qui n'apprécient pas qu' :
- on utilise une variable globale comme argument d'une fonction. Exemple :

```
int Nb_notes;

main
{
    double notes[15];
    ....
    moyenne(notes,Nb_notes);
    ....
}
```

  Beaucoup de compilateurs considèrent que si la variable est globale, elle est toujours visible, donc n'a pas à être passée en argument (thèse qui se défend !). Si vous n'avez pas d'autres solutions, passez par une variable locale intermédiaire.
- on modifie la valeur d'une variable passée par valeur à une fonction. Exemple :

```
double moyenne(double *x, int nb_x)
{
    .....
    nb_x = 15;
    .....
}
```

Certains compilateurs redéfinissent une variable locale nb_x, sans modifier réellement les valeurs passées à la fonction. C'est toutefois une manip très dangereuse, et en tout cas un risque majeur le jour où vous changez de compilateur ou de machine (inutile de décrire les heures à chercher la cause du disfonctionnement !).

– on oublie d'initialiser une zone mémoire. Réserver sa place en mémoire, que ce soit via une allocation dynamique ou statique, n'initialise pas la case mémoire à zéro, comme beaucoup le croit. Attention ! Certains debuggers le font à votre place ! Pensez y lorsqu'un programme marche sous debugger mais pas sans ! ! !

Tous les compilateurs détestent qu'

– on déborde des limites fixées pour un tableau. Ça paraît évident mais il est utile de le vérifier. C'est une des plus grandes sources d'erreurs, en particulier lors de la modification de programmes existants. En géneral on est prudent lors de la première écriture, moins lors d'une modification. Et ne dites pas, « ça a marché » car selon l'emplacement choisi en mémoire lors d'une précédente exécution, le problème a effectivement pu vous être masqué !
– on utilise des indices de 1 à n pour parcourir un tableau ! Quant on alloue un emplacement de n valeurs, les indices se baladent de 0 à $n-1$!

2.13.2 Un résultat de calcul inattendu

Malgré toutes vos vérifications, le résultat d'un calcul n'est pas celui attendu. Vérifiez alors que :

– vous avez bien prototypé la fonction qui est sensé retourner le calcul (oubli d'un fichier include par exemple). Cas très fréquent : vous voulez calculer le sinus d'une valeur et vous codez :

```
main()
{
   a = sin(1.6);
}
```

Si vous n'avez pas au préalable inclus le fichier <math.h> qui définit la fonction sin comme renvoyant un double, le compilateur considèra le résultat comme entier et a vaudra 0.00000 au lieu de 0.999574. Ce qui est valable pour les fonctions mathématiques du C, l'est aussi pour les vôtres !

– vous avez bien passé les arguments en radian et non en degré dans les fonctions trigonométriques.
– vous initialisez bien vos variables. Par défaut, une variable contient n'importe quoi et surtout pas zéro !

Chapitre 3

Compléments de cours

3.1 Le fichier Makefile

Lorqu'un programme a été écrit dans un langage évolué (quelqu'il soit), avant de pouvoir l'exécuter, il faut passer par deux étapes, que nous avons déjà vues : la compilation et l'édition de lien. Ces deux étapes peuvent être grandement simplifiées grâce à l'utilisation d'un fichier makefile qui, s'il est bien rédigé, va tout enchaîner à votre place.

make est donc un outil de maintenance de fichiers. Il travaille à partir d'une liste de fichiers et de la description de leurs relations les uns par rapport aux autres. Lorsqu'on l'appelle, *make* ne génére (ou regénére) que les fichiers strictement nécessaires, en se basant sur les règles de dépendance que vous aurez établies et sur les dates de dernière modification des fichiers concernés. C'est simple : les fichiers objets, par exemple, dépendent de fichiers source et include. Un executable, même simple, est lié aux fichiers objets et aux bibliothèques. Dès qu'un fichier source est modifié, le fichier objet doit être reconstruit et réinséré dans le nouvel exécutable. À partir d'un nombre conséquent de fichiers source, cela devient vite ingérable sans utiliser *make*.

La description des fichiers à maintenir doit être stockée dans un fichier nommé **makefile** ou **Makefile**. Cette description est faite selon une syntaxe particulière qui constitue un mini-langage. De façon générale, la syntaxe est la suivante :

```
< nom du fichier a obtenir > : < fichier dont il depend > < fichier dont il depend > ...
                    < regle pour obtenir le fichier >
```

Soit par exemple :

```
# makefile pour le programme pgm

pgm : s1.o s2.o
        cc s1.o s2.o -o pgm
s1.o : s1.c s1.h
        cc -c s1.c
s2.o : s2.c
        cc -c s2.c
```

La première ligne indique que le fichier exécutable *pgm* dépend des deux fichiers s1.o et s2.o, la troisième ligne que s1.o dépend de s1.c et de s1.h, la cinquième ligne que s2.o dépend uniquement de s2.c.

pgm dépend de s1.o et de s2.o signifie que si la date de dernière modification du fichier *pgm* est postérieure aux deux dates de dernière modification de s1.o et s2.o, tout est correct. Dans le cas contraire, si l'un des deux fichiers a été modifié, *pgm* n'est plus à jour. La ligne suivante indique la commande à lancer pour remettre à jour le fichier *pgm*. Dans ce cas il faut refaire l'édition de lien des deux objets s1.o et s2.o

pour générer un nouvel exécutable *pgm*. Il suffit pour cela de taper *make* sur la ligne de commande. *make* va lire le fichier makefile contenant votre séquence de commande et générer automatiquement la sortie désirée.

Cet outil est particulièrement intéressant pour développer, maintenir et installer des logiciels mettant en oeuvre plusieurs fichiers (sources, librairies, exécutables, doc,...)

Attention de bien commencer la ligne de commande par une tabulation et non une série d'espace. Ce programme est chatouilleux sur ce point !

3.2 Le debugger

Le debugger est un outil vital pour le programmeur ! Il ne s'agit pas d'écrire un programme qui se compile, mais un programme qui marche ! Si les erreurs de compilation sont faciles à corriger (les compilateurs sont d'ordinaire assez explicites sur les causes de l'erreur), il est plus difficile d'appréhender les erreurs de conception logique.

Un debugger est un programme intéractif qui permet de suivre pas par pas, le déroulement d'un programme. Tous les debuggers gèrent les possibilités suivantes :

- **point d'arrêt :** il détermine la ligne de code sur laquelle vous voulez que le programme s'interrompe. Cette instruction ne sera pas exécutée. Les debuggers acceptent qu'il s'agissent d'une ligne spécifiée numériquement (exple : s'interrompre à la ligne 551 du fichier source foo.c) où bien d'un point d'entrée de fonction (exple : s'arrêter au début de la fonction $filter\_gauss()$).
- **exécution en pas à pas :** à partir d'un point d'arrêt, vous pouvez choisir de continuer directement jusqu'au prochain point d'arrêt, ou bien d'exécuter en pas à pas, c'est à dire instruction par instruction, un certain nombre de lignes de code. Vous pouvez aussi choisir de descendre dans le corps d'une fonction ou bien de l'exécuter comme une instruction insécable.
- **affichage et modification de contenu de variable :** vous pouvez à tout moment vérifier le contenu d'une variable, d'un tableau ou d'une structure, et même en modifier le contenu. Certains debuggers sont plus graphiques que d'autres pour la visualisation des données. La dernière génération peut afficher sous forme d'une courbe le contenu d'un tableau.
- **affichage du code source :** selon le niveau d'interfaçage du debugger, vous disposez : soit d'un affichage simple de la ligne de code qui sera la prochaine à s'exécuter (précédée de son numéro de ligne), soit d'une fenêtre avec ascenseur affichant les lignes de codes du fichier source actuellement concerné, où est mise en évidence, par un surlignage, la prochaine ligne de code qui sera exécutée.

Attention ! Pour pouvoir utiliser le debugger, vous devez avoir au préalable compilé vos sources avec l'option de compilation adéquate (usuellement **-g**), sinon vous obtiendrez au lancement un message d'erreur du style « No symbols found ».

3.3 Les outils de debug spécifiques à l'allocation dynamique

Il existe des produits gratuits apportant une aide précieuse lorsqu'on traque un problème d'allocation dynamique. Leur principe de fonctionnement est de substituer aux bibliothèques standard leurs routines propres d'allocation, qui effectueront tout une batterie de tests de cohérence. Tout cela bien sûr au prix d'une certaine lenteur d'exécution, mais pour traquer l'erreur, ils sont irremplaçables. Les plus connus sont :

- $efence$, l'un des plus ancien. Le package contient un fichier d'include et une bibliothèque. Il faut inclure le fichier $.h$ au minimum dans le programme principal, et intégrer la bibliothèque dans l'édition de lien. Plus il y aura de sources contenant le fichier d'include, plus la détection d'erreur sera précise
- $dbmalloc$. Mêmes principes qu'$efence$, mais plus pertinent dans ses détections
- $yamd$, le dernier né (version actuelle 0.32). Son utilisation diffère un peu des deux autres. Vous devez substituer son compilateur au vôtre, puis exécuter le programme à l'aide de run-yamd. Il apparaît à ce jour comme le plus efficace dans ses détections de conflit mémoire.

3.4 Interfaçage avec du code Fortran

Si vous devez interfacer un programme C avec une bibliothèque de fonctions Fortran, vous devez savoir assurer le passage de paramètres d'un langage à l'autre. La première règle est que tous les arguments doivent être passés par **adresse**. Le type de l'argument à passer dépend du tableau d'équivalence ci-après :

Type Fortran	Type C correspondant
INTEGER*1 BYTE	char
INTEGER*2	short int
INTEGER*4	long int
REAL REAL*4	float
REAL*8 DOUBLE PRECISION	double
REAL*16	
COMPLEX COMPLEX*8	structure of 2 floats
COMPLEX*16 DOUBLE COMPLEX	structure of 2 double
COMPLEX*32	
LOGICAL*1	unsigned char
LOGICAL*2	unsigned short
LOGICAL*4	unsigned long
CHARACTER	char
CHARACTER*n	char[n]
Array	array
Sequence derived type	structure

Pour les chaînes de caractères, si la taille n'est pas fixée au préalable par la fonction, le Fortran rajoute, au moment du passage de paramètre, à la fin de la liste d'arguments, la taille de la chaîne. S'il y en a plusieurs, il rajoute les différentes tailles suivant l'ordre de passage des argument de type chaînes. Il faut donc simuler cela à partir du C. Attention, la taille des chaînes, est une donnée passée **par valeur !**

Chapitre 4

Annexes

4.1 Fonctions mathématiques du C

Fonctions inclues dans <math.h>. Tous les arguments x et y sont des double, n, des entiers, et le résultat retourné est toujours un **double**.

```
sin(x)       sinus de x
cos(x)       cosinus de x
tan(x)       tangente de x
asin(x)      arcsinus de x
acos(x)      arccosinus de x
atan(x)      arctangente de x
atan2(y,x)   arctangente de y/x
sinh(x)      sinus hyperbolique de x
cosh(x)      cosinus hyperbolique de x
tanh(x)      tangente hyperbolique de x

exp(x)       exponentielle de x
log(x)       log neperien de x
log10(x)     log en base 10 de x
pow(x,y)     x a la puissance y
sqrt(x)      racine carree de x
fabs(x)      valeur absolue

ceil(x)      plus petit entier superieur ou egal a x
floor(x)     plus grand entier inferieur ou egal a x

ldexp(x,n)   x multiplie par (2 a la puissance n)

frexp(x,*n)  separe x en une fraction normalisee dans l'intervalle [0.5,1[, qui est
             retournee, et une puissance de 2, qui est placee dans *n

modf(x,*y)   separe x en ses parties entiere et fractionnaire, toutes deux du meme
             signe que x. Cette fonction place la partie entiere dans *y et retourne
             la partie fractionnaire.

fmod(x,y)    reste de x/y, exprime en virgule flottante, de meme signe que x
```

4.2 Fonctions de manipulation de chaînes de caractères

Fonctions disponibles dans <string.h>
- int strlen($chaine$)
  retourne la longueur de la chaîne ($\backslash 0$ non compris)
- char *strcpy(char *$destination$, char *$source$)
  recopie la chaîne $source$ dans la chaîne $destination$, et renvoie un pointeur sur le résultat
- char *strncpy(char *$destination$, char *$source$, int n)
  identique à $strcpy()$ mais s'arrête au $\backslash 0$ ou au plus au bout de n caractères (qui doit comprendre le $\backslash 0$)
- char *strcat(char *$destination$, char *$source$)
  concatène la chaîne $source$ à la chaîne $destination$, renvoie un pointeur sur le résultat
- char *strncat(char *$destination$, char *$source$)
  concatène au plus n caractères de la chaîne $source$ à la chaîne $destination$, renvoie un pointeur sur le résultat
- int strcmp(char *$str1$, char *$str2$)
  renvoie 0 si $str1$==$str2$, <0 si $str1$<$str2$, >0 si $str1$>$str2$ selon l'ordre défini par le code ASCII
- int strncmp(char *$str1$, char *$str2$, int n)
  identique à $strcmp()$ mais la comparaison ne porte que sur les n premiers caractères
- char *strchr(char *$chaine$, char c)
  retourne un pointeur sur la première occurence de c dans la chaîne, ou bien NULL si c n'y figure pas
- char *strrchr(char *$chaine$, char c)
  retourne un pointeur sur la dernière occurence de c dans la chaîne, ou bien NULL si c n'y figure pas
- int strspn(char *$chaine$, char *$list\_car$)
  retourne la taille du segment initial de $chaine$ constitué uniquement de caractères appartenant à $list\_car$
- int strcspn(char *$chaine$, char *$list\_car$)
  retourne la taille du segment initial de $chaine$ constitué uniquement de caractères n'appartenant pas à $list\_car$
- char *strpbrk(char *$chaine$, char *$list\_car$)
  retourne un pointeur sur la première occurence de l'un des caractères de $list\_car$ dans la chaîne $chaine$, NULL si aucun n'y figure
- char *strstr(char *$chaine1$, char *$chaine2$)
  retourne un pointeur sur la première occurence de la chaîne $chaine2$ dans la chaîne $chaine1$, NULL si elle n'y figure pas
- char *strtok(char *$chaine$, char *$delimiteurs$)
  recherche et retourne itérativement les lexèmes de $chaine$ délimités par l'un des caractères de la chaîne $delimiteurs$. Le premier appel se fait avec $chaine$ différent de NULL, les suivants avec $chaine$ égal à NULL. On itère le processus jusqu'à ce que la fonction retourne NULL

4.3 Fonctions d'accès mémoire

Fonctions inclus dans <mem.h> :
- int memcmp(void *s1, void *s2, int n)
  compare les n premiers octets des zones mémoires $s1$ et $s2$. Même codes retour que $strcmp()$
- void *memcpy(void *dest, void *src, int n)
  copie les n premiers octets de la zone mémoire src dans $dest$
- void *memmove(void *s1, void *s2, int n)
  idem $memcpy()$ mais fonctionne aussi si les objets se chevauchent
- void *memchr(void *s, char c, int n)
  retourne un pointeur sur la première occurence de c dans s, ou NULL si celui ci n'y figure pas
- void *memset(void *s, char c, int n)
  remplit les n premiers octets de c avec s

La longueur des zone mémoire est toujours donnée en octets. Pensez à utilisez $sizeof()$.

4.4 Options principales de gcc

Seules sont listées ici les options les plus utilisées de gcc. La commande **man gcc** vous en fournira une liste complète.

Syntaxe :

```
gcc [option] filename
```

-c compile seulement les sources. Ne procède pas à l'édition de lien.
-E stoppe après avoir invoqué le préprocesseur. Le résultat est affiché à l'écran.
-o spécifie le nom du fichier de sortie (nom de l'exécutable).
-v affiche les commandes exécutées pour procéder aux différentes étapes de la compilation.
-C demande au préprocesseur de ne pas ôter les commentaires (en utilisation conjointe avec -E).
-l*nom_lib* spécifie à l'éditeur de lien d'utiliser la librairie lib*nom_lib*.a
-Idir spécifie le répertoire dans lequel sont stockés les fichiers d'include. Cette option peut apparaître plusieurs fois, s'il y a plusieurs répertoires
-Ldir spécifie le répertoire dans lequel sont stockés les librairies. Cette option peut apparaître plusieurs fois, s'il y a plusieurs répertoires
-Wall demande au compilateur d'afficher tous les $warnings$ (mises en garde), en plus des erreurs de compilation
-g produit des informations utilisées par le debugger pour la mise au point des logiciels
-O optimise le code pour qu'il soit plus rapide