Langage C - Document de cours

François Goudail
Sylvie Lebrun
Pierre Lugan
Felix Kroeger
Carlos Garrido-Alzar

Table des matières

Introduction

Ce document présente les bases du langage C. Ce sera votre document de référence pendant les séances de travaux dirigés. Il ne contient pas une description exhaustive des fonctionalités de ce langage. Pour une description plus complète, vous pourrez vous référer à la bibliographie (p. 65).

Voici comment utiliser ce document :

- le chapitre 1 décrit brièvement l'interface du compilateur C que vous allez utiliser.
- Le chapitre 2 explique comment réaliser votre premier programme en C.
- Le chapitre 3 vous explique comme programmer de façon correcte et comment diagnostiquer **vous-mêmes** les erreurs de programmation les plus courantes. Certains points ne vous seront pas immmédiatement compréhensibles car ils nécessitent des notions qui seront vues au fur et à mesure du cours. C'est donc un document auquel il vous faudra vous référer périodiquement.
- Les chapitres 4 à 10 décrivent les différents éléments du langage C qui seront abordés au cours des travaux dirigés.

Chapitre 1

Rapide prise en main de Visual C^{++} 2008 express

Visual C^{++} 2008 express est un environnement de développement intégré (EDI) qui allie un logiciel de traitement de texte, un compilateur (programme qui traduit un programme source en langage machine) et qui permet de gérer facilement des projets. Voici une présentation du logiciel avec les principaux éléments qui vous permettront de créer rapidement vos premiers programmes.

1.1 Création d'un fichier source

- Pour débuter, allez dans le menu *fichier*, choisissez *nouveau* puis *projet*. Choisissez comme type de projet *Win32* puis comme modèle *application console win32*.
- Dans *nom*, tapez le nom du projet. Prenez comme habitude de l'appeler par le nom du td, donc **td1** pour la séance 1. Créez un répertoire pour la solution : c'est votre espace de travail. Appelez-le par le nom du binôme (**binome**), vous le réutiliserez à chaque séance. Une solution peut contenir plusieurs projets.
  NB : vous pouvez voir dans l'explorateur de fichiers que l'extension d'une solution est .sln, celle d'un projet est .vcproj
- Pour les paramètres de l'application, choisissez *projet vide*.
- Vous avez ainsi créé la Solution **binome**, avec un projet (**td1**), que l'on peut voir dans l'explorateur de solutions (dans la fenêtre en haut à gauche). Il existe plusieurs types de fichiers dans un même projet : en-têtes, ressources, sources, que l'on peut également voir dans l'explorateur de solutions.
- Pour créer un fichier source, cliquez-droit sur *fichiers sources* dans l'explorateur de solutions, faites *ajouter un nouvel élément*, **forcez l'extension en .c** (par exemple, pour le premier exercice, tapez exo1.c). Ceci permettra d'éviter les conflits de compilation entre les .c et les .cpp. Tapez votre programme dans la fenêtre qui s'ouvre.

1.2 Compilation, édition de liens et exécution

- Une fois le programme écrit, compilez les fichiers source un par un pour détecter les erreurs de syntaxe (dans le menu *générer*, puis *compiler*).
- On peut faire différents types de génération, selon les dépendances d'un projet. Cette année, on ne gère pas de dépendance entre projets, on peut donc utiliser au choix *générer*

+ le nom du projet, ou bien dans le menu générer *projet uniquement* (*générer* permet de compiler tout le projet, de faire l'édition de lien et de créer l'exécutable).
- Pour l'exécution du programme, allez dans le menu *déboguer*, choisissez ensuite *exécuter sans déboggage*.

1.3 Modularité inter-fichiers

A partir de la séance 2, vous allez devoir utiliser des fichiers source que vous aurez créés dans différents projets. Il y a plusieurs façons de s'y prendre, en voici une. Par exemple, dans le projet **td1**, j'ai créé un fichier source **tableaux.c** avec le fichier d'en-tête correspondant **tableaux.h**. Je veux utiliser une fonction définie dans ce fichier dans le fichier **exo2.c** du projet **td2**.
Je clique-droit sur le projet **td2** dans l'explorateur de solutions et sélectionne *ajouter élément existant*. Je sélectionne **tableaux.c** et **tableaux.h**. Dans le fichier **exo2.c**, j'inclus au programme le chemin d'accès au fichier d'en-tête par la ligne de commande

```
#include " chemin-d'accès-au fichier d'entête.tableaux.h ".
```

L'intérêt de cette méthode est que l'on a un seul fichier **tableaux.c** que l'on peut mettre en commun avec plusieurs projets, donc si on fait une modification dans ce fichier, elle se fera automatiquement pour tous les projets. L'inconvénient est que le chemin d'accès peut être différent d'un ordinateur à l'autre, d'où une limitation de la portabilité.

Chapitre 2

Un premier programme

Dans ce chapitre nous présentons et commentons un premier exemple simple de programme en C.

2.1 Un programme simple

Voici le programme :

```
/* Affiche bonjour */
#include<stdio.h>
main()
{
    printf("bonjour\n");
}
```

Voici quelques commentaires sur l'organisation de ce programme :

- Le programme débute par des commentaires placés entre les symboles /* et */. Il s'agit de textes explicatifs destinés aux lecteurs du programme et qui n'ont aucune incidence sur la compilation. Même s'ils peuvent être placés à n'importe quel endroit, il vaut mieux choisir des emplacements propices pour la lisibilité. Notez que si les commentaires s'étendent sur une seule ligne il est possible de ne mettre que les symboles // en début de ligne.
- La première ligne du programme `#include<stdio.h>` s'appelle une directive à destination du préprocesseur, c'est-à-dire qu'elle est prise en compte avant la compilation du programme. Ces directives débutent toujours par le symbole #, sont écrites à raison d'une seule par ligne et sont généralement placées en début de programme. Attention, il n'y a jamais de point-virgule à la fin d'une directive. Ici, la directive demande d'introduire des instructions (en langage C) situées dans le fichier `stdio.h` qui vont servir au programme. Il existe d'autres sortes de directives que nous verrons plus loin.
- `main` s'appelle la fonction principale du programme. Il n'y a qu'une et une seule fonction principale dans un programme (l'exécution débute par cette fonction) et son nom est toujours `main`. On verra qu'à part celà la fonction `main` se comporte comme les autres fonctions. Ce que fait la fonction `main` est délimité par les accolades { et }.
  L'ensemble des instructions délimitées par ces accolades s'appelle un bloc.
- Dans notre premier exemple, il n'y a qu'une seule instruction : `printf("bonjour\n");`
  Cette instruction affiche bonjour à l'écran grâce à la fonction `printf` qui est déjà définie

dans le fichier `stdio.h` (et donc vous n'avez pas à l'écrire vous-même) et effectue ensuite un saut à la ligne grâce au symbole conventionnel `\n`.
Notez bien qu'en C, toutes les instructions se terminent par un point-virgule.

2.2 Conduite à tenir en cas de messages d'erreur

Voici quelques conseils lorsqu'on rencontre des messages d'erreur :

- En cas d'erreur, le programme nous avertit par un message dans la fenêtre du bas. On peut avoir une idée de l'emplacement de l'erreur en double-cliquant sur le message : une flèche bleue apparaît dans le fichier source, à la ligne de l'erreur. Les erreurs les plus fréquentes et les moyens de les diagnostiquer vous même à partir du message d'erreur sont expliqués en Section 3.4.
- Il arrive souvent qu'on obtienne un grand nombre de messages d'erreurs, alors que le programme contient une seule erreur qui se répercute à plusieurs endroits. **Donc, quand vous "débuggez" un programme, il faut toujours commencer par corriger la première erreur**.
- Des erreurs à l'édition de liens peuvent survenir et sont souvent dues à des erreurs dans les chemins d'accès.
- Il peut également y avoir des "warnings" qui sont souvent de simples avertissements (perte de précision dans la conversion d'un nombre par exemple). Même si on peut faire l'édition de liens avec des warnings à la compilation, il faut tout de même les surveiller pour être sûr qu'ils ne cachent pas quelque chose de plus "grave".

Chapitre 3

"Bonnes pratiques" de programmation

Ce chapitre est destiné à vous donner quelques lignes directrices pour une bonne programmation et à répondre à quelques interrogations que vous rencontrerez à un moment ou un autre du cours de programmation en C. Il prend la forme d'une liste de conseils à suivre et d'erreurs à éviter, fondée sur l'expérience des années précédentes.

3.1 Règles générales pour le développement de programmes

3.1.1 En résumé

Un code doit fonctionner correctement Il doit donc avoir été testé dans toutes les configurations d'utilisation possibles (voir le point 3.1.2).

Un code doit être réutilisable par une personne autre que celle qui l'a programmée. Toute fonction doit donc être correctement documentée. Une bonne documentation doit contenir

- Une description de l'opération réalisée par la fonction
- La description des paramètres d'entrée et de sortie
- La date de dernière modification et le nom de celui qui l'a modifié

Dans les entreprises, il existe des procédures très sévères permettant d'assurer la documentation et la "réutilisabilité" du code. Dans ce cours de base, nous ne les mettrons pas en œuvre. Mais il est bon d'acquérir dès maintenant de bonnes habitudes, qui vous seront très utiles lors de votre première expérience en entreprise ou dans un laboratoire de recherche (où ces notions sont tout aussi utiles).

A plus brève échéance, n'oubliez pas qu'au cours de cet enseignement de C, vous aurez à réutiliser des fonctions que vous aurez programmées lors des séances précédentes. Donc prenez soin de la qualité de votre documentation !

Un code doit pouvoir être modifié par une personne autre que celle qui l'a programmée Le code doit donc être suffisammment commenté. Bien entendu, il n'est pas besoin de mettre une ligne de commentaire par ligne de code, mais en ayant toujours à l'idée que votre code doit être compris par une autre personne, vous saurez trouver le juste milieu (les points 3.1.3 et 3.1.4 vous proposent une approche).

Il faut noter que la documentation d'une fonction et le commentaire du code sont deux choses différentes, **qui sont aussi indispensables l'une que l'autre** :

- La documentation permet à quelqu'un d'utiliser facilement la fonction sans avoir à lire le code.
- Les commentaires permettent à une personne qui veut modifier le code de le faire facilement.

3.1.2 Testez vos fonctions

Testez vos fonctions avec **plusieurs jeux de paramètres** pertinents et songez aux possibles cas pathologiques : division par zéro, débordement si tel paramètre prend telle valeur etc...Vos premières applications n'ont pas vocation à répondre à des cahiers des charges draconiens ; il ne s'agit donc pas de rattraper tous les cas pathologiques possibles, mais de songer un minimum à ceux que vous pourriez rencontrer dans l'emploi ultérieur de vos fonctions.

3.1.3 Commentez *à bon escient* le corps des fonctions

Il existe diverses écoles pour ce qui est de l'usage des commentaires dans les fonctions. **Une fonction doit être suffisamment simple** (voir le point 3.1.6), utiliser des noms de variables suffisamment explicites, et recourir suffisamment peu au « bidouillage », pour pouvoir se passer de commentaires dans son corps. Dans les fonctions qui ne peuvent faire l'économie d'une certaine longueur, commentez la fonction des différents blocs logiques. Commentez aussi les points délicats. De façon générale, les commentaires ne doivent pas nuire à la lisibilité du code !

3.1.4 Commentez *très précisément* les fonctions en tête de leur définition

Ce commentaire qui figure devant vos fonctions est le cahier des charges de votre fonction. Il doit être précis pour qu'un tiers utilisateur (ou vous-même une semaine plus tard) puisse **utiliser en toute confiance votre fonction, sans être obligé de faire de l'ingénierie à rebours**. Ces commentaires devraient en principe être reproduits dans les fichiers d'en-têtes, puisque ceux-ci demeureront lisibles en texte clair même si vous distribuez vos programmes sous forme compilée.

Exemple **Vous avez écrit une fonction qui tire une suite d'entiers aléatoires équirépartis dans l'intervalle** (a,b), et vous voulez maintenant utiliser ces entiers comme indices de tableau. Nous voudriez-vous pas exactement savoir si les bornes a et b de l'intervalles sont incluses ou exclues, c'est-à-dire si l'intervalle est fermé ou ouvert, à gauche et à droite ? A défaut vous risquez soit de rencontrer de graves problèmes de débordement de tableaux, soit de ne jamais voir sortir du tirage aléatoire les indices des éléments extrémaux du tableau. Dans les deux cas, vous perdrez beaucoup de temps à découvrir l'origine du problème. **L'inclusion ou l'exclusion de a et de b du tirage doit être spécifiée dans le commentaire en tête de la définition de la fonction qui effectue le tirage.**

3.1.5 Séparez calcul et affichage dans des fonctions distinctes

Pendant la phase de développement d'une fonction qui effectue un calcul, il peut être utile d'afficher la valeur de certaines variables intermédiaires, par exemple si votre fonction ne marche pas et que vous voulez remonter à la source de l'erreur. Néanmoins, quand vous êtes certains que votre fonction effectue correctement ce que vous lui demandez, éliminez les affichages de votre fonction. Cela évitera les affichages intempestifs lors de la réutilisation de la fonction. Cela vous

forcera à retourner réellement les grandeurs calculées en variables de sortie de la fonction, sachant qu'**une valeur simplement affichée à l'écran est a priori perdue pour le programme principal**. Réservez l'affichage de valeurs ou leur exportation vers un fichier à des fonctions qui remplissent spécifiquement ces tâches.

Pratique Gardez cette recommandation à l'esprit quand vous répondez à énoncé succinct du genre : « Ecrivez un programme [une fonction] qui calcule x et l'affiche à l'écran. » Il s'agit alors
- d'écrire une première fonction qui calcule x à partir d'autres quantités et renvoie x en variable de sortie,
- d'écrire une seconde fonction qui prend en entrée une variable du même type que x et l'affiche,
- et d'employer les deux fonctions dans un programme principal ou une autre fonction qui réponde à l'énoncé.

De la même façon, **bannissez au maximum les interactions avec l'utilisateur** lorsque vos fonctions sont suffisamment «mûres».

Exemple Lorsque vous écrivez une fonction qui enregistre des données dans un fichier, ne recourez pas au *prompt* (`scanf`) pour demander le nom de fichier à l'utilisateur pendant l'exécution de la fonction, mais fournissez au contraire ce nom de fichier à la fonction par un argument d'entrée, sous forme d'une chaîne de caractères. C'est le programme principal qui se chargera éventuellement de l'interaction avec l'utilisateur.

3.1.6 Subdivisez les tâches

Dans l'esprit du point qui précède, subdivisez les tâches. Une fonction simple est facile à écrire (commentaires compris), facile à tester, facile à relire plus tard, et présente l'avantage d'être **générique**, c'est-à-dire plus souvent réutilisable qu'une fonction qui réalise une tâche complexe et par conséquent très spécifique. N'oubliez pas qu'en réutilisant vos fonctions, **vous diminuerez votre quantité de travail**.

3.1.7 Notion d'entrées/sorties de fonctions

La notion d'entrée et de sortie de fonction n'est pas tant liée à la distinction entre les arguments (entre parenthèses) de la fonction et ce qui figure derrière le mot-clé return (rien ou une *unique* donnée en C !). Une entrée est une variable ou adresse qui à l'entrée de la fonction contient de l'information significative pour la fonction. Une sortie est une variable ou adresse qui à la sortie de la fonction contient de l'information qu'elle ne contenait pas avant. Un pointeur peut donc être à la fois entrée et sortie.

3.1.8 Ecriture de bibliothèques

Comment organiser les fichiers source et regrouper les fonctions :
- Mettre toutes les fonctions dans des bibliothèques, c'est à dire des fichiers `.c` où les fonctions sont regroupées par thème (une bibliothèque pour les fonction de lecture/ecriture de fichiers, une autre pour les fonctions liées aux statistiques, pour les fonctions de tri, etc.).
- Créer un fichier d'en-tête (fichier `.h`) pour chaque bibliothèque.

– Maintenir une documentation de toutes les fonctions de la bibliothèque. A la fin du cours, vous devrez fournir vos bibliothèques de programmes documentées et commentées.

3.2 Modèle de fonction à utiliser

Voici le modèle de fonction qu'il vous sera demandé d'utiliser systématiquement dans vos programmes :

```
/* dicho : recherche de la racine d'une équation par dichotomie
   ENTREE :
   - a,b : intervalle de recherche - On suppose que f(a)*f(b)<0
   - eps : précision dela solution trouvée
   - f : fonction dont on veut touver la racine
   SORTIE :
   - racine de l'équation
**** F. Goudail - 27/06/06
*/

double dicho(double a, double b, double eps, double (*f) (double))
{
    double x;

    // On se place dans le cas où f(a)<0
    if ((*f)(a) > 0) {x=a;a=b;b=x;}

    // Début de la boucle
    while (fabs(b-a)>eps)
    {
        x=(a+b)/2;
        if ((*f)(x)<0)
            a=x;
        else

            b=x;
    }

    return x;
}
```

3.3 Quelques remarques plus spécifiques au C

3.3.1 Comment renvoyer deux variables ?

A éviter : stocker ces deux variables dans un tableau et renvoyer le tableau. Cette approche est lourde, maladroite si les deux variables ont des significations différentes (l'âge du capitaine et le nombre de coté d'un dodécagone), et devient impraticable si les deux variables sont des types

différents. **A faire : passer ces deux variables par adresse (pointeurs) en arguments à la fonction.** A éviter aussi : la solution bâtarde qui consisterait à renvoyer une deux variables par le return et l'autre par adresse, sauf si cela a un sens bien précis, comme lors de l'utilisation de codes d'erreur. Si vous avez à retourner un ensemble cohérent de données inhomogènes, vous pouvez utiliser une structure (voir chapitre 10).

3.3.2 En-têtes et directives *include*

La première chose que fait le compilateur est l'analyse lexicale de votre code, linéairement, de la première ligne de code à la dernière. S'il tombe sur un mot qu'il ne connaît pas encore, il proteste. Ceci se produit lorsqu'un mot n'est un pas un mot du langage natif (`main`, `double`, `if` ...) ou **qu'une fonction ou une variable de votre cru n'a pas été *déclarée* préalablement, «plus haut» dans votre code** (notez qu'une fonction ou variable peut ne pas encore avoir été *définie* ou *initialisée*, nuance ; les avertissements potentiels sont alors d'une autre nature). C'est pour cela que l'on place en tête de fichier source les *déclarations* ou prototypes des fonctions, dont les *définitions* suivent plus loin, afin que le nom de ces fonctions puisse être employable dans ce fichier même en des endroits qui précèdent la définition de la fonction. C'est aussi le rôle que remplissent les directives `#include` en tête de fichier source : elle copient-collent littéralement le contenu du fichier en argument à l'endroit où apparaît la directive ; ainsi on peut inclure dans un fichier source les fichiers d'en-têtes (`.h`) qui déclarent les fonctions (et constantes) d'autres fichiers sources ou de ce même fichier source.

Pratique Une bonne pratique lors de l'écriture d'un fichier source consiste donc à
- écrire les définitions de fonctions dans le fichier source (`source.c`),
- regrouper parallèlement les déclarations de ces fonctions et les définitions de constantes dans le fichier d'en-têtes correspondant (`source.h`),
- inclure `source.h` dans `source.c` par la directive `#include` avant compilation (à supposer que les fonctions de `source.c` aient besoin les unes des autres, ou qu'elles utilisent des constantes de votre cru).

3.4 Dépannage / *Troubleshooting*

Les messages d'erreur qu'affiche votre compilateur (en bas de votre interface de développement, dans la disposition standard des fenêtres) sont plus explicites qu'il n'y paraît ! Apprenez à vous aider vous-mêmes. Avec l'expérience, vous associerez un type de message d'erreur à un problème spécifique dans votre code, vous saurez ainsi plus rapidement localiser votre erreur et la corriger. Ne négligez pas les avertissements (*warnings*) : parfois ils pointent du doigt des imperfections du code qui amèneront à de véritables erreurs plus tard. Sachez que certains compilateurs sont plus laxistes que d'autres.

3.4.1 unresolved external symbol _WinMain@16 [erreur à l'édition de liens]

```
Linking...
LIBCD.lib(wincrt0.obj) : error LNK2001 : unresolved external symbol _WinMain@16
Debug/win32appError.exe : fatal error LNK1120 : 1 unresolved externals
Error executing link.exe..
```

Explication Vous avez bâti un projet Win32 Application, au lieu de Win 32 Console Application.

Solution Créez un nouveau projet, et recopiez-y vos sources ...

3.4.2 La mémoire ne peut être « written » [à l'exécution dans la console]

Explication Votre programme essaye d'écrire à une adresse (dans une zone de mémoire) non autorisée. Ceci arrive dans les **débordements de tableau** ou lorsque vous mélangez **adresses et variables** comme dans l'exemple suivant :

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

int main ()
{
    int n = 9; // variable de type entier
    //...
    scanf("%d",n); // argument attendu: adresse de type hexadecimal

    return 0;
}
```

Explication `n` doit être une adresse aux yeux de `scanf` dans la syntaxe ci-dessus, il y a donc conversion explicite de l'entier `n` vers sa représentation hexadécimale, et le programme tente d'écrire le résultat de `scanf` à l'adresse `9`. Dans le cas présent, il convient de spécifier l'adresse `&n` à `scanf`.

3.4.3 variable used without having been initialized [avertissement à l'édition de lien, puis erreur « written » si l'on passe outre]

```
int main ()
{
    int *x;
    //...
    *x = 10;

    return 0;
}
```

```
Compiling...
source1.c
E :\home\pierre\infoC2008-2009\FAQws\pointerNotInitialized\source1.c(5) : warning
C4700 : local variable 'x' used without having been initialized
source1.obj - 0 error(s), 1 warning(s)
```

Explication Le pointeur `x` a été déclaré, mais pas initialisé, autrement dit `x` ne pointe encore vers aucune adresse, ou une adresse aléatoire, lorsque l'on essaie d'écrire `10` à cette adresse.

Solution Initialiser `x` par un `malloc` ou une instruction du type `x=y` où `y` est un pointeur déjà initialisé (ou `x=&v` où `v` est une variable déjà déclarée, de type compatible).

3.4.4 illegal indirection [erreur à la compilation]

```
int main ()
{
    int x;
    /...
    *x = 1;

    return 0;
}
```

ou bien encore

```
int main ()
{
    int x,y;
    //...
    y = *x;

    return 0;
}
```

```
Compiling...
source1.c
E :\home\pierre\infoC2008-2009\FAQws\illegalIndirection\source1.c(5) : error C2100 : illegal
indirection
Error executing cl.exe.
source1.obj - 1 error(s), 0 warning(s)
```

Explication L'*indirection* est l'usage de l'opérateur `*` pour récupérer la valeur vers laquelle pointe un pointeur (*dereferencing*) ; ici, `x` n'est pas un pointeur, l'utilisation de l'opérateur `*` sur `x` est donc inappropriée.

Solution Vérifiez l'homogénéité des éléments que vous manipulez à une variable, un pointeur, un pointeur de pointeurs, etc.

3.4.5 '=' : left operand must be l-value [erreur à la compilation]

```
int main()
{
    double x,y;
    double *z;
    //..
    z = &x; // ok
    &y = &x; // warning
```

```
    return 0;
}
```

```
Compiling...
source1.c
E :\home\pierre\infoC2008-2009\FAQws\leftOperand\source1.c(7) : error C2106 : '=' : left
operand must be l-value
Error executing cl.exe.
source1.obj - 1 error(s), 0 warning(s)
```

Explication `y` a été déclarée comme variable, son adresse ne peut être changée.

Solution Déclarez `y` comme pointeur, initialisez-le par `malloc`, l'adresse vers laquelle il pointe peut alors être changée.

```
#include<stdlib.h>

int main()
{
    double array[2];
    array = malloc(2*sizeof(double));

    return 0;
}
```

```
Compiling...
source2.c
E :\home\pierre\infoC2008-2009\FAQws\leftOperand\source2.c(6) : error C2106 : '=' : left
operand must be l-value
Error executing cl.exe.
source2.obj - 1 error(s), 0 warning(s)
```

Explication `array` a été déclaré comme tableau, son adresse ne peut être changée.

Solution Déclarez `array` comme pointeur, puis initialisez-le par `malloc`. Son adresse peut ensuite être changée. Remarquez ici la limite de l'analogie entre tableaux et pointeurs.

3.4.6 incompatible types [avertissement à la compilation]

```
int main()
{
    double x=0.;
    int i;
    int *j;

    i=x;
    j=&x;
```

```
    return 0;
}
```

```
Compiling...
source1.c
E :\home\pierre\infoC2008-2009\FAQws\incompatibleTypes\source1.c(7) : warning C4244 :
'=' : conversion from 'double ' to 'int ', possible loss of data
E :\home\pierre\infoC2008-2009\FAQws\incompatibleTypes\source1.c(8) : warning C4133 :
'=' : incompatible types - from 'double *' to 'int *'
source1.obj - 0 error(s), 2 warning(s)
```

ou bien encore

```
int main()
{
    int i=0;
    double x;
    double *y;

    x=i;
    y=&i;

    return 0;
}
```

```
Compiling...
source2.c
E :\home\pierre\infoC2008-2009\FAQws\incompatibleTypes\source2.c(8) : warning C4133 :
'=' : incompatible types - from 'int *' to 'double *'
source2.obj - 0 error(s), 1 warning(s)
```

Explication Remarquez la difference de comportement du compilateur dans les quatre opérations d'affectation ci-dessus. Dans le cas des variables, la conversion implicite de type suscite tout au plus un avertissement lorsqu'il y a perte de précision (`i=x`). Lorsque l'on travaille avec des pointeurs, par contre, l'incompatibilité des types de données sous-jacents (`int` et `double`) provoque des avertissements dans les deux cas de figure.

Solution Il est recommandé d'éviter les conversions implicites dans les quatre cas, de les rendre explicites (*casting*) si besoin, et de les documenter. Evitez toutefois d'avoir recours au *casting* de façon systématique, uniquement pour faire disparaître les avertissements. Il y a peut-être un problème sous-jacent qui ne sera pas résolu par cet artifice. Vous rencontrerez ce genre de situations lorsque vous tenterez, par exemple, de réutiliser avec des réels une fonction que vous avez écrite pour des entiers ; prenez alors le temps de réécrire la fonction pour des réels.

3.4.7 Où vérifier l'homogénéité de mes fonctions/données ?

Nombre des erreurs répertoriées ci-dessus (*incompatible types, illegal indirection, operator must be l-value* ...) sont dues à l'inhomogénéité des types que vous mettez en regard dans une instruction. Les programmes sont rarement aussi triviaux que les exemples ci-dessus, et il vous faudra parcourir votre code à la recherche des occurrences des fonctions ou données (pointeurs

ou variables) stigmatisées par le compilateur, car l'erreur ne se situe pas toujours à la ligne indiquée dans le message d'erreur, mais parfois en amont, dans un fichier d'en-tête, dans un autre programme source que vous avez modifié à la hâte (par exemple en changeant le prototype d'une fonction) sans répercuter les modifications dans les autres fichiers sources du même projet, etc.

Pratique Pensez-donc à vérifier la cohérence des types entre

- les différents endroits où la fonction/donnée apparaît dans la présente source,
- le fichier en-tête où la fonction/donnée apparaît,
- le fichier source (potentiellement différent de celui que l'on est en train de compiler) qui définit la fonction/donnée,
- les autres fichiers sources qui emploient la fonction/donnée.

Si l'erreur se produit non pas à la compilation, mais à l'édition de liens, il est probable que l'inhomogénéité soit partagée entre plusieurs fichiers source.

Chapitre 4

Eléments de base du langage C

4.1 Les variables

Le langage C, comme beaucoup de langages, utilise des variables, c'est-à-dire des emplacements en mémoire qui portent un nom et dont le contenu est susceptible d'évoluer. L'instruction `int i ;` est la déclaration de la variable qui a pour nom i. `int` est le type de i. `int` est un mot-clé, c'est-à-dire un mot réservé par le langage pour un usage bien défini et qui ne peut pas être utilisé autrement (voir la liste des mots-clés en annexe A.1). `int` permet de réserver un emplacement en mémoire d'une certaine taille pour la variable i et de coder cette variable d'une certaine façon (nous verrons comment plus loin). L'instruction `i=1 ;` permet d'initialiser la variable i à 1.
Les noms des variables (ou identificateurs) sont formés d'une suite de caractères choisis parmi les lettres ou les chiffres, le premier d'entre eux étant nécessairement une lettre. Ces noms seront choisis de façon judicieuse pour la lisibilité du programme.

4.2 Les types simples

La mémoire de l'ordinateur est un ensemble de bits, regroupés en octets (8 bits), chaque octet étant repéré par ce qu'on appelle son adresse. L'ordinateur ne sait représenter et traiter que des informations exprimées sous forme binaire. Dans la machine, toute information (nombre entier, nombre réel, instructions, caractères ...) est codée sous cette forme. On voit alors par exemple que le motif binaire 01001101 peut être interprété différemment selon sa nature : il peut s'agir du nombre entier 77 ou du caractère M. D'où l'importance de la notion de *type*, qui indique en particulier la manière dont l'information a été codée.
Les types de base du langage se répartissent en trois catégories, que nous détaillons dans la suite.

4.2.1 Les types entiers

Le C prévoit que, sur une machine donnée, on puisse trouver 3 tailles différentes d'entiers, désignées par les mots-clés suivants :

- short
- int (c'est le plus utilisé)
- long int

Chacun de ces types correspond à une taille de données (en nombre d'octets) précise. Toutefois ces tailles de données dépendent de la machine utilisée : tous les int n'ont pas la même taille sur toutes les machines. Fréquemment, deux des trois mots-clés correspondent à une même taille.

Chacun des trois types peut être nuancé par l'utilisation du qualificatif `unsigned` (non signé). Il n'y a alors plus de bit pour coder le signe et on ne représente plus que des nombres positifs. Le tableau donné ci-dessous récapitule les différentes manières de spécifier chacun des types (les plus courants) ainsi que la taille minimale et le domaine minimal imposés par la norme, quelle que soit l'implémentation concernée.

Spécificateurs de type possible	Taille minimale (octets)	Domaine minimal
short	2	-32 767 à +32 767
unsigned short	2	0 à 65 535
int	2	-32 767 à +32 767
unsigned int	2	0 à 65 767
long, long int	4	-2 147 483 647 à +2 147 483 647

4.2.2 Les types flottants

Les types flottants permettent de représenter, de façon approchée, une partie des nombres réels. En C, il y a trois types de flottants correspondant à des tailles différentes :

- float : réel simple précision
- double : réel double précision (c'est le plus utilisé)
- long double (précision étendue)

Le tableau ci-dessous donne, sur un système de taille moyenne, la taille en octets, les valeurs maximales et minimiales ainsi que la précision des différents types flottants.

Spécificateurs de type	Taille (octets)	Domaine (positif)	Précision
float	4	$3,4 \times 10^{-38}$ à $3,4 \times 10^{+38}$	7
double	8	$1,7 \times 10^{-308}$ à $1,7 \times 10^{+308}$	15
long double	10	$3,4 \times 10^{-4932}$ à $3,4 \times 10^{+4932}$	19

4.2.3 Le type caractère

Le type caractère correspond au mot-clé `char`. Le C permet de manipuler des caractères codés en mémoire sur un octet, la plus petite partie adressable de la mémoire. Par ailleurs la notion de caractère en C dépasse celle de caractère imprimable. Il existe ainsi certains caractères de passage à la ligne (comme on l'a déjà utilisé dans l'exemple du chapitre 2), de tabulation ... Nous donnons ci-dessous les principales combinaisons appelées *séquences d'échappement* :

\n	nouvelle ligne
\t	tabulation
\b	espace-arrière
\r	retour-chariot
\f	saut de page
\a	signal sonore

Une constante de type caractère se note en écrivant le caractère entre apostrophes (par exemple, 'A'). Une constante chaîne de caractères se note en écrivant ses caractères entre guillemets (par exemple, "bonjour").

Dans la machine, chaque caractère est codé sur un octet selon un code appelé *code ASCII* (voir annexe A.2). La valeur d'une constante caractère est le nombre qui représente le code ASCII du caractère.

4.3 Conversion de types

Avant de pouvoir calculer une expression, toutes les données doivent être converties dans un même type. La plupart de ces conversions se passent automatiquement, sans l'intervention du programmeur, mais il est important de prévoir leur effet. Parfois il est nécessaire de convertir une donnée dans un type différent de celui que choisirait la conversion automatique ; dans ce cas, nous devons forcer la conversion à l'aide d'un opérateur spécial ("cast").
Si un opérateur a des opérandes de différents types, les valeurs des opérandes sont converties automatiquement dans un type commun. Ces manipulations implicites convertissent en général des types plus "petits" en des types plus "larges" ; de cette façon on ne perd pas en précision.
Considérons l'exemple suivant :

```
int x=5 ;
float y=2.5 ;
float z=y/x ;
```

Pour le calcul de `z`, la variable `x` est d'abord convertie en type `float` puis la division est effectuée. Le résultat est de type `float` et vaut 0.5.
En revanche, lors d'une affectation, la donnée à droite du signe d'égalité est convertie dans le type à gauche du signe d'égalité. Dans ce cas, il peut y avoir perte de précision si le type de la destination est plus faible que celui de la source.
Après les instructions suivantes :

```
int x=5 ;
float y=2.5 ;
x=y ;
```

la variable `x` vaut 2.
Il est possible de convertir explicitement une valeur en un type quelconque en forçant la transformation comme dans l'exemple suivant :

```
char A=3 ;
int B=4 ;
float C ;
C = (float)A/B ;
```

La conversion se fait par l'opérateur de cast `(float)`. La valeur de A est explicitement convertie en *float*. La valeur de B est automatiquement convertie en *float*. Le résultat de la division (type rationnel, valeur 0,75) est affecté à C. Le résultat est `C=0.75`.
Il faut cependant faire très attention aux parenthèses lorsqu'on utilise l'opérateur de `cast`. En effet, considérons la suite d'instructions suivantes :

```
char A=3 ;
int B=4 ;
float C ;
C = (float)(A/B) ;
```

Le résultat est `C=0`. En effet le résultat de 'A divisé par B' est de type `int` (le plus "grand" des types de A et B) et vaut 0. Ce résultat est ensuite converti en flottant et vaut donc toujours 0.

NB : l'opérateur de `cast` *est à utiliser avec une grande prudence, il vaut mieux l'éviter lorsque c'est possible.*

Notez qu'en C, le quotient de deux entiers est un entier. Ainsi 1/4 vaut 0. En revanche le quotient de deux flottants est bien un flottant : 1.0/4.0 vaut bien 0.25.

4.4 Les opérateurs

4.4.1 Les opérateurs arithmétiques

Le langage C dispose d'opérateurs classiques "binaires" (c'est-à-dire portant sur deux opérandes) : l'addition (+), la soustraction (-), la multiplication (*) et la division (/) ainsi que d'un opérateur "unaire" (c'est-à-dire ne portant que sur un seul opérande) correspondant à l'opposé et noté -. Les opérateurs binaires ne sont définis que pour deux opérandes ayant le même type et ils fournissent un résultat de même type que leurs opérandes. Si les deux opérandes sont de types flottants différents, une troncature sera réalisée sur le flottant de plus grande précision.
Il existe un opérateur modulo noté % qui ne peut porter que sur des entiers et qui fournit le reste de la division euclidienne du premier opérande par le second.

4.4.2 Les opérateurs relationnels

Le C permet de comparer des expressions avec les opérateurs habituels > , < , <=, >=, != (*différent de*) et l'opérateur d'égalité ==. *ATTENTION ! L'opérateur = est réservé à l'affectation d'une valeur à une variable, ce qui est différent.*

Le résultat d'une comparaison est un entier qui vaut 0 si le résultat de la comparaison est faux, 1 si le résultat de la comparaison est vrai. Une comparaison devient alors une expression de type entier et pourra éventuellement intervenir dans des calculs arithmétiques.

4.4.3 L'opérateur d'affectation

L'instruction `i=5 ;` réalise une action : l'affectation de la valeur 5 à la variable i. L'opérateur d'affectation = possède une priorité inférieure à celle de tous les opérateurs arithmétiques et de comparaison si bien que dans l'instruction `c=b+5 ;` l'expression b+5 est d'abord évaluée puis le résultat est affecté à c.

4.4.4 Les opérateurs logiques

Il y a trois opérateurs logiques classiques : et (&&), ou (||) et non (!). Par exemple (a<b)&&(c<d) prend la valeur 1 (vrai) si les deux expressions (a<b) et (c<d) sont toutes deux vraies (de valeur 1) la valeur 0 (faux) dans le cas contraire.

4.4.5 Les opérateurs d'incrémentation et de décrémentation

Le C dispose d'un opérateur unaire noté ++ qui permet de réaliser des pré ou post incrémentations d'une variable selon qu'il est placé avant ou après cette variable. L'opérateur – joue un rôle symétrique sur la décrémentation.
Dans l'exemple suivant :

```
a=5 ;
```

`b=++a-5 ;`
La variable `a` est d'abord incrémentée, puis ensuite on lui soustrait `5`. La variable `b` prend donc la valeur 1. La variable `a` prend la valeur 6.

En revanche, dans l'exemple suivant :
`a=5 ;`
`b=-5+a++ ;`
on soustrait d'abord `5` à `a`. La variable `b` prend donc la valeur 0. La variable `a` est ensuite incrémentée et prend la valeur `6`.

ATTENTION ! la syntaxe de ces exemples doit être comprise car, elle apparaît souvent dans les programmes en C, mais **vous ne devez pas les utiliser** car elles rendent les programmes illisibles. Dans le cadre de ce cours, la seule utilisation "autorisée" sera `a++` ou `a--`.

4.5 Les instructions de contrôle

4.5.1 L'instruction for

L'instruction `for` sert à faire des boucles. Sa syntaxe est la suivante :

for (expression_1 ;expression_2 ;expression_3) instructions ;

expression_ 1 initialise la variable compteur de la boucle. *expression_ 2* est le test de continuation qui compare cette variable et une valeur limite : si le résultat est nul, l'exécution de la boucle *for* est terminée. Sinon on exécute le bloc *instructions* et on évalue *expression_ 3*, qui consiste à incrémenter le compteur. On refait alors le test de continuation et ainsi de suite.
Voici un exemple d'utilisation de l'instruction *for* :

```
int i ;
for (i=0 ; i<10 ; i++)
{
    printf("%d\n" , 2*i);
}
```

Dans cette boucle, on fait varier `i` de 0 à 9.

4.5.2 L'instruction if

L'instruction *if* permet de programmer une structure dite de choix, permettant de choisir entre deux instructions, suivant la valeur d'une expression numérique jouant le rôle de condition. Sa syntaxe est la suivante :

if (expression) instructions1 ;
else instructions2 ;

La seconde partie, introduite par le mot-clé *else*, est facultative.
Cette instruction évalue l'expression mentionnée à la suite de *if*. Si elle est non nulle, on exécute le bloc instructions1. Si elle est nulle, on exécute le bloc instructions2 si ce bloc est présent. Puis, dans tous les cas, on passe à l'instruction suivant cette instruction *if*.
Cas des if imbriqués : un *else* se rapporte toujours au dernier *if* rencontré (dans le même bloc)

auquel un *else* n'a pas encore été attribué. L'utilisation systématique de blocs, même réduits à une instruction simple, évite d'avoir à se poser la question du rattachement d'un *else*.
Voici un exemple d'utilisation de *if* :

```
if (a == 1)
{
    if (b == 1)
        printf("bonjour\n");
}
else
    printf("salut\n");
```

4.5.3 L'instruction switch

Cette instruction permet de faire des choix multiples. Voici sa syntaxe :
switch(expression)
{
case constante_1 : [suite d'instructions_1]
case constante_2 : [suite d'instructions_2]
...
case constante_n : [suite d'instructions_n]
[default : suite d'instructions]
} *expression* est une expression de type entier. *constante_i* est une expression constante entière. *suite d'instructions_i* est une séquence d'instructions quelconques. Les crochets [et] signifient que leur présence est facultative.
Lors de l'exécution, l'expression est évaluée et il y a branchement à l'étiquette *case xxx* correspondante si elle existe ; toutes les instructions qui la suivent sont alors exécutées, jusqu'à la fin du bloc ou jusqu'à une instruction de rupture `break`. Dans le cas contraire, il y a branchement à l'étiquette *default* si elle existe, à la suite de l'instruction *switch* sinon.
Voici un exemple d'utilisation de l'instruction *switch* :

```
int i = 1;
char a ;
scanf("%c" , &a);
switch(a)
{
    case 'o' : i++ ;
                        printf("%d\n",i);
                        break;
    case 'n' : i-- ;
                        printf("%d\n",i);
                        break;
    default :  printf("%d\n",i);
}
```

Dans le morceau de programme ci-dessus, si le caractère entré au clavier est 'o', le programme affiche 2, si le caractère est 'n', le programme affiche 0, sinon, il affiche 1.

4.5.4 Les instructions do while et while

La structure de *do...while* est la suivante :

do instructions ; while (expression) ;

Cette instruction exécute le bloc *instruction* puis elle évalue l'*expression* de contrôle suivant les règles habituelles : si cette dernière est nulle, elle passe à l'instruction suivant *do ...while* et l'exécution de la boucle est terminée ; dans le cas contraire (expression non nulle), on reprend la suite des instructions du bloc *instructions*.
La structure de *while* est la suivante :

while (expression) instructions ;

Cette instruction évalue l'*expression* de contrôle suivant les règles habituelles : si cette dernière est nulle, elle passe à l'instruction suivant *while* et l'exécution de la boucle est terminée ; dans le cas contraire (expression non nulle), on reprend la suite des instructions du bloc *instructions*.
Voici un exemple d'utilisation de l'instruction *while* :

```
int i;
i=1;
while (i>1)
{
    printf("%d\n",i);
    i--;
} //on ne rentre pas dans la boucle, le programme n'affiche rien
```

Voici un exemple d'utilisation de l'instruction *do ... while* :

```
int i;
i=1;
do
{
    printf("%d\n",i);
    i--;
} while (i>1) ; //on rentre 1 fois dans la boucle, le programme affiche 1
```

4.6 Les directives au précompilateur

En langage C, la traduction d'un fichier source se déroule obligatoirement en deux étapes indépendantes : un prétraitement et une compilation proprement dite. Le programme qui réalise la première étape s'appelle le préprocesseur ou précompilateur, le second s'appelant le compilateur. Le travail du précompilateur consiste en un traitement du texte du fichier source fondé sur l'interprétation d'instructions particulières, structurées en lignes et qu'on appelle des directives. Nous allons voir dans la suite les trois types de directives que nous utiliserons le plus cette année.

4.6.1 Les inclusions de fichiers

Il s'agit de la directive `#include` qui permet d'incorporer dans le fichier source un texte provenant d'un autre fichier qu'on nomme généralement un fichier en-tête. Le contenu de ce

fichier vient alors remplacer la directive `#include` et il est analysé par le préprocesseur, comme s'il avait été directement placé à cet endroit dans le fichier source. Notez qu'on a déjà vu cette directive avec `#include<stdio.h>` qui permettait d'utiliser dans le programme principal des fonctions de la bibliothèque standard contenues dans le fichier `stdio.c` (comme la fonction printf).

4.6.2 Les constantes symboliques

Une directive telle que :
```
#define NMAX 10
```
demande de remplacer la constante symbolique `NMAX` par le texte 10 à chaque fois que ce symbole apparaîtra dans la suite du fichier source. **Attention à ne jamais mettre de point-virgule à la fin d'une ligne de directive.**

4.6.3 Les macroinstructions

La directive suivante permet de calculer le carré d'une variable :
```
#define SQR(x) ((x)*(x))
```
Le symbole à substituer se présente sous la forme d'un idenficateur (SQR) suivi d'un paramètre entre parenthèses (x). La directive demande au préprocesseur de remplacer dans la suite du fichier source tous les textes de la forme SQR(x), dans lesquels x représente en fait un texte quelconque, par((x)*(x)). Par exemple `sqr(12)` deviendra 12*12.
Dans la pratique, on évitera d'utiliser ce genre de directives car elles sont induisent un grand nombre d'erreurs. Il vaut mieux utiliser des fonctions, comme on va le voir par la suite.

4.7 Les entrées-sorties formatées

4.7.1 La fonction printf

Utilisation

Considérons l'exemple suivant :
```
int i=5 ;
printf ("le carré de %d est %d",i,i*i) ;
```
Cette suite d'instructions affiche la phrase suivante : "le carré de 5 est 25".
On voit ainsi que la fonction printf peut prendre trois arguments (on verra plus loin la définition d'un argument mais on peut retenir qu'ici il s'agit d'un paramètre d'entrée de la fonction). Le premier argument est une chaîne de caractères comprise entre les guillemets. Dans cette chaîne de caractères, certains sont à afficher tels quels : *le carré de est.* Des codes de format repérés par le symbole % permettent de préciser au compilateur un code de conversion (d dans l'exemple) donnant le type de l'information à afficher. Les deux arguments suivants sont les valeurs que l'on souhaite afficher à la place des codes de format.

Les principaux codes de conversion de printf sont donnés ci-dessous :

c	char
d	int
u	unsigned int
ld	long
lu	unsigned long
f	double ou float (compte tenu des conversions systématiques de float vers double) écrit en notation décimale avec 6 chiffres après le point
e	double ou float (compte tenu des conversions systématiques de float vers double) écrit en notation exponentielle avec 6 chiffres après le point, sous la forme x.xxxxxxe+yy ou x.xxxxxxe-y et-x.xxxxxxey ou -x.xxxxxxe-y pour les nombres négatifs
g	double ou float (compte tenu des conversions systématiques de float vers double) choisit entre la notation %f ou %e celle qui est la plus "élégante". Permet également de supprimer les zéros inutiles après la virgule.
s	chaîne de caractères dont on fournit l'adresse
x	hexadécimal

Par défaut, les entiers sont affichés avec le nombre de caractères nécessaire. Les flottants sont affichés avec six chiffres après le point. Il est possible de modifier ces valeurs par défaut.

Précision de l'affichage

On peut demander à préciser l'affichage grâce à des éléments à rajouter dans le format de *printf* après le caractère %, par exemple :

- - : cadrer à gauche du champ (par défaut, le cadrage est à droite)
- + : imprimer le signe du nombre (même positif)
- *espace* : si le 1er caractère n'est pas un signe, placer un espace au début d'un nombre
- 0 : compléter les nombres par des zéros (à la place des blancs)

On peut ajouter aussi un nombre qui indique la largeur du champ. Il s'agit d'une largeur minimum, automatiquement augmentée si elle est insuffisante.
Voici une liste d'appels de `printf` avec l'affichage produit par chacun, après les instructions `float x; x = 123.456;` :

```
printf(">%f<",x);        >123.456001<
printf(">%12f<",x);      >  123.456001<
printf(">%12.2f<",x);    >  123.46<
printf(">%.2f<",x);      >123.46<
printf(">%-12.2f<",x);   >123.46      <
printf(">%+-12.2f<",x);  >+123.46     <
printf(">%012.2f<",x);   >000000123.46<
```

4.7.2 La fonction scanf

La fonction scanf permet à l'utilisateur de rentrer une information au clavier et au programme de stocker cette information dans une variable. Voici un exemple d'utilisation de la fonction scanf :

```
double x;
printf("entrez un nombre au clavier :");
scanf("%lf",&x);
```

Ce morceau de programme permet de stocker dans la variable x la valeur saisie au clavier par l'utilisateur. Notez bien la présence du symbole & devant le x (on verra que ce symbole permet d'accéder à l'adresse de la variable x).

Pour lire une donnée, la fonction `scanf` a besoin de qu'on lui spécifie le format de cette donnée. Les principaux codes de conversion sont donnés ci-dessous :

c	char
d	int
u	unsigned int
hu	unsigned short
ld	long int
lu	unsigned long
f ou e	float écrit indifféremment dans l'une des deux notations
lf ou le	double avec la même présentation que ci-dessus
s	chaîne de caractères dont on fournit l'adresse

Notez aussi l'absence du caractère d'échappement \n dans le motif passé entre guillemets à la fonction `scanf`. Il faudrait sinon deux appuis sur la touche entrée pour terminer la saisie.

4.8 Les fonctions getchar et putchar

getchar est une fonction (en réalité une macroinstruction) qui lit les données d'une zone tampon du fichier d'entrée standard *stdin* (le clavier le plus souvent) et qui fournit les données seulement après confirmation par 'Enter'. Les valeurs retournées par *getchar* sont soit des caractères, soit le symbole EOF (qui signifie *End Of File* pour marquer la fin d'un fichier). Comme EOF n'est pas un caractère, le type de la valeur retournée par *getchar* est *int*. Voici un exemple d'utilisation de *getchar* :

```
int c ;
c=getchar() ; // Retourne le code ASCII du caractère entré au clavier
```

L'instruction `putchar('a') ;`transfère le caractère a vers le fichier standard de sortie *stdout* (en général l'écran). Voici des exemples d'utilisation de *putchar* :

```
char A = 100;
putchar(68);   // affiche le symbole dont le code ASCII est 68  ('D')
putchar(A);    // affiche le symbole dont le code ASCII est 100 ('d')
```

4.9 Redirection de flots

A ce stade, il peut être utile d'écrire les résultats qu'affichent vos programmes à l'écran dans un fichier texte. Il existe une façon simple d'écrire et de lire un tel fichier à partir de commandes sous DOS (on verra plus tard comment le programmer en C).
Pour écrire les résultats dans un fichier, sous l'invite de commandes, il suffit de se placer dans le répertoire où se trouve votre fichier exécutable qu'on appellera `programme.exe` par exemple et de taper la commande suivante :

```
U:\...programme.exe > resultats.txt
```

Le fichier `resultats.txt` va être ainsi créé dans le répertoire de travail et les résultats de votre programme seront à la fois écrits à l'écran et dans ce fichier. Pour lire les données d'un fichier déjà existant (`datas.txt` par exemple) et en faire des entrées de votre programme, sous l'invite de commandes, il suffit de se placer dans le répertoire où se trouve `programme.exe` et de taper la commande suivante :

```
U:\...programme.exe < datas.txt
```

4.10 Les tableaux à une dimension

Comme tous les langages, le C permet d'utiliser des tableaux. Il s'agit d'un ensemble d'éléments de même type désignés par un identificateur unique et rangés côte à côte en mémoire ; chaque élément est repéré par un indice précisant sa position au sein de l'ensemble. Supposons que l'on souhaite utiliser un tableau de vingt entiers. Il faut tout d'abord le déclarer avec l'instruction suivante : `int tab[20] ;`
Le nom du tableau *tab* est donné par le programmeur. Notez bien que la taille du tableau, ici 20, est **obligatoirement une constante** (un nombre ou une constante symbolique).

De façon conventionnelle, en C, le premier élément d'un tableau est repéré par l'indice 0 et le dernier par l'indice de la taille moins 1 : dans notre exemple, le premier élément du tableau tab est tab[0] et le dernier est tab[19].
Attention aux problèmes de débordement, tab[20] n'existe pas : le programme peut soit faire n'importe quoi, soit envoyer un message d'erreur pas toujours très clair.

4.10.1 Initialisation d'un tableau

Il est possible d'initialiser un tableau en utilisant une boucle :

```
int i;
for (i = 0 ; i<20 ; i++) tab[i]=i; //initialise chaque composante du tableau tab
                                   //à la valeur de son indice
```

Lors de la déclaration d'un tableau, on peut aussi initialiser les composantes du tableau, en indiquant la liste des valeurs respectives entre accolades :
`int A[5] = {10, 20, 30, 40, 50};`
Il faut bien veiller à ce que le nombre de valeurs dans la liste corresponde à la dimension du tableau. Si la liste ne contient pas assez de valeurs pour toutes les composantes, les composantes restantes sont initialisées à zéro.

4.11 Les chaînes de caractères

Une constante de type chaîne de caractères représente une suite finie de caractères, de longueur quelconque. Les caractères constituant la chaîne sont rangés côte à côte en mémoire, dans l'ordre où ils figurent dans la chaîne. Un caractère nul est ajouté immédiatement après le dernier caractère de la chaîne pour en indiquer la fin. Par caractère nul, on entend le caractère dont le code interne est 0. On peut le noter indifféremment 0, '000' ou '0'.

On verra au chapitre 9 comment manipuler ces chaînes de caractères. Notez déjà qu'on déclare une chaîne de caractères comme un tableau de type char : `char chaine[20] ;`.

Chapitre 5

Les fonctions

5.1 Modularité et consignes de programmation

Comme tous les langages, le C permet de découper un programme en plusieurs parties nommées souvent "modules". Cette programmation modulaire se justifie pour de multiples raisons parmi lesquelles la lisibilité des programmes, le partage d'outils communs, la hiérarchisation des idées.
En C, il n'existe qu'un seul type de module : la fonction. Vous devrez utiliser au maximum des fonctions dans vos programmes. Un programme sera jugé "de bonne qualité" lorsqu'il contiendra les fonctions que vous aurez réalisées et que la fonction principale `main` contiendra uniquement des instructions faisant appel à ces fonctions. Il faudra veiller à ce que la lecture seule de la fonction `main` suffise à comprendre la logique de votre programme.
Nous allons voir dans la suite comment réaliser et utiliser une fonction, puis nous étudierons un exemple concret.

5.2 Déclaration et définition d'une fonction

Une fonction est, comme son nom l'indique, un module qui *fait* quelque chose. Une fonction peut avoir des paramètres d'entrée qu'on appelle des arguments (mais celà n'est pas obligatoire) et peut renvoyer une valeur (mais celà n'est pas obligatoire non plus). Pour utiliser une fonction, il y a plusieurs règles à respecter :

La déclaration

Afin de compiler un fichier source utilisant une certaine fonction, le compilateur a seulement besoin de savoir qu'il va rencontrer, quelque part dans le programme, la définition de cette fonction, c'est-à-dire ce qu'elle fait. Cette information lui est donnée par la déclaration de la fonction, qui donne au compilateur les éléments nécessaires à la compilation.
Losque votre programme ne comporte qu'un seul fichier source, la déclaration de la fonction se fait généralement au début du programme, avant la fonction `main` et après les directives du préprocesseur. Lorsque votre programme comporte plusieurs fichiers source, la déclaration d'une fonction se fera dans le fichier en-tête .h que vous aurez créé et qui contiendra l'ensemble des fonctions de votre programme ayant la même thématique.
La déclaration d'une fonction consiste à donner son en-tête (ou prototype), par exemple :

```
double poly(double x, int a, int c);
```

poly est le nom de la fonction. *x, a et c* s'appellent les arguments formels de la fonction, c'est-à-dire qu'on aurait très bien pu les appeler autrement : z, coeff1 et coeff2 par exemple. On aurait tout aussi bien pu ne pas les mettre car ce qui est important au niveau de la déclaration d'une fonction, c'est d'indiquer le type des variables. On aurait donc pu déclarer la fonction de la manière suivante :

```
double poly(double , int , int ) ;
```

Toutefois le fait de préciser le nom des variables ne coûte rien et rend le programme plus lisible. Le terme *double*, en début de définition, indique que la fonction *poly* renvoie une valeur et que cette valeur est de type double. Enfin, il ne faut pas oublier le point-virgule à la fin de la déclaration.
Voici un autre exemple de déclaration de fonction :

```
void affichage() ;
```

affichage est une fonction qui ne prend pas d'argument (il ne faut cependant pas oublier les parenthèses indiquant qu'il s'agit d'une fonction) et qui ne renvoie pas de valeur (ce qui est précisé par le terme *void*).

La définition

La définition de la fonction, c'est-à-dire ce qu'elle fait, doit se placer en dehors de toute autre fonction. On la place généralement après la fonction `main` ou dans un autre fichier source qui contient des fonctions sur la même thématique.
La définition de la fonction *poly* est :

```
//poly calcule la valeur d'un polynôme du second degré
double poly(double x, int m, int n)
{
    double val ;
    val = x*x + m*x + n ;
    return val ;
}
```

Notez qu'il est utile d'indiquer au début de la fonction ce qu'elle fait, grâce à des commentaires. On verra d'ailleurs au paragraphe 3.2 la façon dont nous vous demandons de commenter vos fonctions.
x, m et n sont toujours les arguments formels de la fonction. *val* est une variable *locale* de la fonction, elle n'existe qu'à l'intérieur de la fonction *poly*. L'instruction *return val* indique que *val* est la valeur retournée par la fonction. Lorsqu'on utilise des fichiers .c, il est préférable de toujours utiliser l'instruction *return* dans toute fonction, même si la fonction n'a pas de valeur (cet oubli qui ne coûte qu'un warning avec des .c génère par contre une erreur à la compilation avec des .cpp, donc autant prendre de bonnes habitudes tout de suite).
La définition de la fonction *affichage* est :

```
//affiche une ligne de * à l'écran
void affichage()
{
    printf("******************\n") ;
    return;
}
```

La fonction *affichage* ne prend pas d'argument et ne renvoie aucune valeur (d'où l'instruction *return* suivie d'un point-virgule). Par contre, elle a une action : elle affiche une ligne de * à l'écran.

L'appel d'une fonction

L'appel d'une fonction se fait en nommant la fonction et en remplaçant les arguments formels par des arguments effectifs (sans donner leur type), par exemple :

```
double y = 0 ;
int x = 1 ;
int z = 3 ;
poly(y, x, z) ;
```

y, x et z sont des arguments effectifs. La valeur de *poly(y, x, z)* est 3.
L'appel de la fonction *affichage* se fait comme suit :

```
affichage() ;
```

NB : la définition d'une fonction peut se faire après son appel. En revanche la déclaration d'une fonction doit se faire **obligatoirement avant** le premier appel.

5.3 Argument tableau dans une fonction

Il est souvent utile de passer un tableau comme argument d'une fonction. Voici par exemple la fonction `initialisation` qui initialise les composantes d'un tableau à 0 :

```
void initialisation(int tab[],int dim) //la dimension dim du tableau est passée en argument
{
    int i;
    for(i=0;i<dim;i++)
        tab[i]=0;
}
```

Pour initialiser un tableau `tab2` déclaré par :

```
int tab2[100] ;
```

il suffit d'écrire l'instruction :

```
initialisation(tab2,100) ; //on donne uniquement le nom du tableau sans les crochets
et sa taille
```

5.4 Portée et durée de vie des variables

Les déclarations de variables peuvent se trouver :

- à l'intérieur d'un bloc (ensemble d'instructions délimité par des accolades), il s'agit alors de variables **locales**. Une variable locale ne peut être utilisée que depuis l'intérieur du bloc où elle est déclarée. En aucun cas on ne peut y faire référence depuis l'extérieur de ce bloc. Dans le bloc où il est déclaré, le nom d'une variable locale masque toute variable de même nom définie dans un bloc englobant le bloc en question.
  Pour faciliter la lisibilité des programmes, il est fortement conseillé de déclarer toutes les variables locales ***au début*** *de la fonction.*

- dans l'en-tête d'une fonction, il s'agit alors d'arguments formels. Les arguments formels d'une fonction sont considérés comme des variables locales déclarées au début du bloc le plus extérieur. En aucun cas on ne peut y faire référence depuis l'extérieur de la fonction.
- en dehors de toute fonction, il s'agit alors de variables **globales**. Le nom d'une variable globale peut être utilisé depuis n'importe quel point compris entre sa déclaration et la fin du fichier où la déclaration figure, sous réserve de ne pas être masquée par une variable locale ou un argument formel de même nom.

Les variables locales et les arguments formels des fonctions ont un espace alloué en mémoire le temps du bloc ou de la fonction en question, cet espace est rendu au système à la fin de ce bloc ou de cette fonction. Les variables globales existent quant à elles pendant toute la durée de l'exécution. Lorsque nous étudierons des programmes contenant plusieurs fichiers source, nous verrons qu'il existe des qualifieurs (comme `static` ou `extern`) qui régissent la visibilité inter-fichiers des variables globales.

Chapitre 6

Modularité et fichiers

Jusqu'à présent, les programmes que vous avez réalisés contenaient une ou deux fonctions et étaient écrits dans un seul fichier source. Vos programmes vont maintenant s'étoffer et il est désormais indispensable d'utiliser la modularité inter-fichiers que permet le langage C. Cette modularité présente de nombreux avantages :

- Elle permet d'une part de hiérarchiser les problèmes à résoudre.
- D'autre part il est possible en C de compiler séparemment les différents fichiers source d'un même projet. Ceci permet un gain de temps appréciable lorsqu'il s'agit de gros fichiers. De plus une erreur dans un fichier ne se répercute pas sur les autres fichiers.
- Un autre avantage de la modularité est qu'elle permet également l'échange de fichiers. Dans la suite de votre travail vous devrez **obligatoirement utiliser la modularité, c'est à dire décomposer vos programmes en fonctions**. On va voir dans les paragraphes suivant la façon de procéder.

6.1 Fichiers en-tête

La notion de projet prend maintenant toute sa signification puisqu'il y aura désormais dans vos projets plusieurs fichiers source : l'un d'entre eux *exclusivement* contiendra la fonction `main` et chaque autre fichier source du projet aura un thème bien défini. Par exemple, on appellera le fichier concernant des manipulations sur les tableaux `tableaux.c`. Dans ce fichier, on mettra toutes les définitions de fonctions concernant les manipulations de tableaux (fonction d'initialisation, d'affichage ...). On mettra aussi toutes les directives d'inclusion de fichiers dont le compilateur a besoin pour compiler ce fichier de façon autonome.
Il faut ensuite créer un fichier en-tête .h (pour *header*) qui contiendra toutes les déclarations (prototypes) des fonctions du fichier `tableaux.c`. Ce fichier s'appellera `tableaux.h`. Dès qu'on voudra utiliser dans un fichier source du projet une fonction définie dans `tableaux.c`, il suffira ainsi d'inclure le fichier `tableaux.h` grâce à la directive `#include"tableaux.h"`. Notez que le nom du fichier s'écrit entre guillemets si le fichier se trouve dans le répertoire courant (s'il est écrit entre $<$ et $>$, le compilateur va chercher dans une liste de chemins d'accès définie au paramétrage du logiciel).

6.2 Modularité et variables globales

Pour rendre une variable globale à un seul fichier source (et donc non visible par les autres fichiers source), il faut la déclarer `static` au début du fichier où elle est utilisée, avant toute

fonction. Par exemple, dans le fichier `tableaux.c`, la déclaration `static int a ;` rendra la variable `a` visible par toutes les fonctions de ce fichier uniquement.
Pour rendre une variable globale visible par plusieurs fichiers du projet, il faut la déclarer `extern` au début de l'un des fichiers, puis sans qualificatif dans les autres. Par exemple, si on déclare `extern double b ;` au début du fichier `tableaux.c` et `double b ;` au début du fichier contenant la fonction `main`, la variable `b` sera visible par l'ensemble des fonctions des deux fichiers considérés.

Chapitre 7

Les pointeurs

7.1 Allocation statique de mémoire

Pour l'instant, nous avons manipulé des variables (numériques de différents types, tableaux, ...). Les données correspondant à ces variables sont stockées dans la mémoire du processeur. Par exemple, l'ensemble d'instructions

```
double a;
a = 5;
```

définit une variable, et lui donne la valeur "5". Cette valeur est stockée dans la mémoire, dans un emplacement repéré par une adresse. Typiquement, la mémoire d'un ordinateur est divisée en "cases" de 8 bits, appelés octets. Ces cases sont rangées séquentiellement. Elles sont traditionnellement repérées par leur code hexadécimal [1]. Ainsi, la case `B10D` suit la case `B10C`

L'utilisateur ne contrôle pas en général l'adresse mémoire dans laquelle sera stocké le contenu d'une variable qu'il définit. Dans l'exemple précédent, le processeur pourra choisir de placer le contenu de la variable `a` à l'adresse `B10C`. Comme cette variable est de type `double`, sa taille est 4 octets. Elle occupera donc les cases mémoire d'adresse `B10C` à `B10F`.

Au cours de l'exécution du programme, le processeur libère l'emplacement mémoire d'une variable lorsque la variable arrive en "fin de vie". Cette "fin de vie" dépend du type de variable :

- **Pour une variable globale** : L'allocation de mémoire a lieu au début de l'exécution du programme, et la mémoire est libérée à la fin de l'exécution du programme.
- **Pour une variable locale** : (on rappelle que c'est une variable interne à une fonction) L'allocation de mémoire a lieu au début de l'exécution de la fonction, et la mémoire est libérée en fin d'exécution de la fonction.

L'allocation statique est pratique car l'ordinateur se charge lui-même d'allouer l'espace mémoire nécessaire lors de la déclaration d'une variable, et de le libérer automatiquement lorsqu'elle arrive en fin de vie. Cependant, dans ce processus, comme la déclaration et l'allocation sont liées, il est indispensable de connaître la dimension de la variable lorsqu'on la déclare.

```
double   a;          // Déclaration et allocation d'un double
int     n;          // Déclaration et allocation d'un int
double   tab[100];   // Déclaration et allocation d'un tableau de double
```

[1] Le code hexadécimal est un mode de numération en base 16, composé des symboles suivants : 0,...,9,A,B,C,D,E,F.

```
double   tab[n];     // Erreur de compilation ! Il faut que la dimension
                     // du tableau soit fixée lorsqu'on le déclare, pour
                     // que l'ordinateur puisse allouer la mémoire.
```

Dans de nombreux cas, on ne connaît pas la taille d'une variable au moment de sa déclaration. Par exemple, dans un programme destiné à traiter une image, l'une des variables sera un tableau contenant l'image. Or en général, on ne connaît pas la taille de l'image qu'aura à traiter le programme car cette taille peut varier. D'autre part, lorsqu'une variable est dupliquée, par exemple, par la commande

```
double a,b; a=5; b=a;
```

la variable `b` est déclarée, la valeur "5" lui est allouée et est stockée à une adresse mémoire différente de celle de `a`. De même, si cette variable apparaît en argument d'une fonction, par exemple

```
fonc(a);
```

une copie de la variable est créée en mémoire, et c'est sur cette copie que va travailler la fonction. Cela peut poser deux types de problèmes :

- Si la variable est volumineuse, par exemple un tableau, ou une image, sa duplication occupe de la mémoire de manière inutile. En effet, pour accéder à la valeur de la variable, il suffirait de connaître son adresse en mémoire.
- A l'intérieur d'une fonction, on travaille sur une copie de la variable. On peut donc pas modifier la variable de départ. La variable `a` ne peut pas être modifiée par la fonction. C'est une transmission de paramètre "par valeur". Or dans de nombreux cas, il est utile que la fonction puisse modifier la valeur d'une variable passée en argument. Pour cela, il faut que la fonction connaisse **l'adresse** de la variable `a`.

Pour toutes ces raisons, le langage C définit des **"pointeurs", qui sont des types de variables spéciaux destinés à contenir l'adresse d'une zone mémoire**. Dans la plupart des programmes en C, il sera nécessaire, ou bien plus pratique de manipuler des pointeurs, qui sont de simples adresses, plutôt que les variables elles mêmes. En particulier, on utilisera les pointeurs dans les cas suivants :

- Pour pouvoir déclarer une variable (en général, un tableau) sans connaître sa dimension. Nous verrons qu'il existe en fait un lien très fort entre la notion de pointeur et celle de tableau.
- Pour qu'une fonction puisse modifier la valeur d'une variable passée en argument. On parle alors de "transfert par adresse".
- Pour passer des variables volumineuses (tableaux, structures) à des fonctions, même si on ne souhaite pas modifier leur valeur. En effet, en manipulant simplement leurs adresses, on n'a pas besoin de dupliquer leur contenu.

7.2 Variables et adresses

Considérons la suite d'instructions suivante :

```
int n; n=5;
```

L'opérateur `&` permet d'avoir accès à l'adresse d'une variable, c'est à dire l'adresse de la première case mémoire dans laquelle est stockée la valeur de la variable `n`.

L'opérateur `*` donne accès à la valeur pointée par une adresse :

```
printf("%d",*(&n));
```

affiche la valeur de la variable `n`, c'est à dire 5.

7.3 Les pointeurs

Un pointeur est une variable dont la valeur est l'adresse d'une case mémoire. C'est une variable **typée**, dont le type est celui de la donnée contenue dans la case mémoire qu'il pointe. Par exemple, la déclaration d'un pointeur sur un entier se fait de la manière suivante :

```
int     *p; // Création d'un pointeur dont le nom est "p" et qui
            // est destiné à pointer sur une donnée entière (int). Pour
            // l'instant, il ne contient aucune adresse
int     x;  // Allocation statique d'un entier
p=&x;       // On définit la valeur du pointeur p, qui est l'adresse de
            // la variable x.
```

Comme le pointeur `p` contient une adresse, on peut accéder au contenu de cette adresse en utilisant l'opérateur `*`. Par exemple,

```
x=5;
printf("%d\n",x)    // Affichage de la valeur de x, c'est à dire 5
printf("%d\n",*p)   // Affichage de la valeur de x, c'est à dire 5
```

Cas particulier important : Le pointeur `NULL` est défini par défaut dans le langage C. Il pointe sur l'adresse `0`. On s'en sert pour indiquer qu'un pointeur n'est pas défini. Par exemple, on verra que la fonction `malloc`, qui alloue un espace mémoire et renvoie un pointeur qui pointe sur cet espace, renvoie le pointeur `NULL` lorsqu'elle n'a pas réussi à allouer l'espace demandé.

7.4 Pointeurs et tableaux

Il important de retenir qu'en langage C, **un tableau est équivalent à un pointeur**. En effet, une déclaration du type

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

crée en fait un pointeur, dont le nom est `tab`, qui pointe sur la case mémoire contenant la première valeur du tableau. Par exemple,

```
for (i=0;i<100,i++)
    tab[i]=i;                    // remplissage du tableau
printf("%d,%d\n",tab[0],*tab)   // donne : 1,1
```

Les autres éléments du tableau sont contenus dans les cases mémoires qui suivent `tab`. Donc en C, **les notions de tableau et de pointeur sont totalement équivalentes**.

7.5 Allocation dynamique de la mémoire

Cependant, pour définir un tableau, il ne suffit pas de déclarer un pointeur. Il faut aussi "allouer" l'emplacement mémoire qui sera réservé aux données contenues dans le tableau. En fait, l'instruction `int tab[100]` fait deux choses :

- Elle déclare un pointeur de type entier, qui pointe sur une certaine case mémoire.
- Elle "alloue" 99 autres cases mémoire, rangées à la suite de la première, qui vont servir à contenir les données du tableau, dont on a fixé le nombre à 100 lors de la déclaration.

L'inconvénient de cette méthode, on l'a dit, est qu'on doit connaître la taille du tableau au moment de sa déclaration. Une autre solution consiste à

- Déclarer un pointeur au début du programme.
- Lui allouer "dynamiquement" l'emplacement mémoire nécessaire au moment du programme où on le connaîtra.

L'allocation dynamique de mémoire permet donc à l'utilisateur d'allouer la quantité de mémoire qu'il souhaite pour stocker des données. L'espace mémoire alloué est une suite de cases mémoire contigües. L'emplacement de cet espace sera repéré par l'adresse de sa première case. Cette adresse sera contenue dans un pointeur.

Pour allouer dynamiquement de la mémoire à un pointeur déjà déclaré, on utilise la fonction

```
malloc(taille)
```

Cette fonction alloue une zone mémoire de dimension `taille`, et elle renvoie l'adresse de la première case de cette zone sous la forme d'un pointeur. Par exemple :

```
#include <stdlib.h>
...
int     dim;
double   *tab;
...
dim=100;
tab=malloc(dim*sizeof(double))
```

Cette suite de commandes permet de créer le tableau `tab` puis de lui allouer un espace mémoire de 100 valeurs de type `double`. On aurait pu aboutir au même résultat avec la commande `int tab[100]`, mais la grosse différence, **c'est que la taille de l'espace mémoire n'a pas besoin d'être connu au moment où la variable `tab` est déclarée**. Elle peut être le résultat d'un calcul ou d'une lecture de données ayant eu lieu au cours du programme. Mis à part cela, le pointeur `tab` est totalement équivalent à un tableau. On peut l'utiliser avec exactement la même syntaxe :

```
for (i=0;i<100,i++) tab[i]=2*i;
printf("%d,%d\n",tab[1],tab[99])    // donne 2,198
```

Nous verrons dans la section suivante qu'on peut utiliser une autre syntaxe pour parcourir les éléments du tableau.

NB1 : L'opérateur `sizeof` retourne la taille d'un type de données. C'est une fonction très pratique car elle évite d'avoir à retenir le nombre d'octets qu'occupe un type de données. C'est d'autant plus utile que le nombre d'octet d'un `int` peut varier selon le processeur utilisé !

NB2 : La fonction `malloc` est contenue dans la librairie `stdlib.h`. Donc si on veut l'utiliser , il faut mettre l'instruction `#include <stdlib.h>` au début du fichier.

7.6 Libération de l'espace mémoire

Remarque importante : Tout espace mémoire alloué avec `malloc` doit être libéré dès qu'on ne s'en sert plus. C'est très important, en particulier lorsque `malloc` est utilisée à l'intérieur d'une fonction. En effet, l'espace n'est pas automatiquement libéré lorsqu'on sort de la fonction. Si cette fonction est utilisée un grand nombre de fois dans le programme, elle va à chaque fois allouer une zone mémoire différente, et si on ne la libère pas avant de sortir de la fonction, on risque rapidement de saturer la mémoire.

La fonction permettant de libérer l'espace mémoire répéré par un pointeur s'appelle `free`. Pour libérer l'espace mémoire pointé par le pointeur `tab`, on utilise :

```
free (tab);
```

7.7 Opération sur les pointeurs

Un certain nombre d'opérations peuvent être appliquées aux pointeurs.

- L'opérateur `+` permet de se déplacer dans la mémoire à partir de l'adresse contenue dans le pointeur. Si `p` est un pointeur sur une donnée de taille `T` octets, `p+5` désigne le pointeur sur l'adresse située `5*T` octets plus loin.
- L'opérateur `-` permet d'aller dans l'autre sens.
- Les opérateurs `++` et `--` permettent de d'incrémenter et de décrémenter un pointeur

Par exemple :

```
double   *tab; tab=malloc(100*sizeof(double));
for (i=0;i<100,i++) *(tab+i)=3*i;
printf("%d,%d,%d\n",*tab,*(tab+1),*(tab+99));  // donne 0,3,297
p=tab+10;printf('%f\n',*(p-5));  // donne tab[5]=15
```

On obtiendrait exactement les mêmes résultats avec la syntaxe suivante :

```
printf("%d,%d,%d\n",tab[0],tab[1],tab[99]);
printf("%d\n",tab[5]);
```

En C, les tableaux et les pointeurs sont donc tout à fait équivalents.

ATTENTION : Il est interdit d'aller écrire dans des zones de la mémoire qui n'ont pas été allouées. Soit par exemple :

```
int   *tab;
tab=malloc(100*sizeof(int))
*(tab+100) = 5;
```

Dans cette dernière commande, on va écrire dans une zone mémoire qui n'a pas été allouée. De deux choses l'une : soit cette zone n'est pas utilisée, et il ne se passe rien. Soit elle est utilisée, et le fait de changer sa valeur va faire "planter" le programme. **L'écriture dans des zones de la mémoire interdites est une source fréquente d'erreurs dans les programmes utilisant des pointeurs**.

7.8 Tableau à deux dimensions

En allocation statique, la syntaxe de déclaration et d'utilisation d'un tableau à deux dimensions est la suivante :

```
int tab[10][20];
tab[0][10] = 1;
```

Un tableau à deux dimensions peut être vu comme un "pointeur de pointeur", c'est à dire un tableau de pointeurs Le pointeur `tab` contient l'adresse de la case mémoire `tab[0]`, qui elle même contient un pointeur sur la case mémoire contenant la valeur `tab[0][0]` du tableau. Les 19 autres valeurs de la première ligne du tableau sont aux adresses `tab[0]+i`, où i varie de 1 à 19. De même, la case mémoire pointée par `tab+1=tab[1]` contient un pointeur sur la case mémoire contenant la valeur `tab[1][0]` du tableau. Les valeurs de la seconde ligne du tableau sont donc contenues dans les case mémoires obtenues en incrémentant le pointeur `tab[1]`. Les autres lignes du tableau sont structurées de la même manière.

En raison de cette structure, l'allocation dynamique d'un tableau à deux dimensions est légèrement plus complexe que celle d'un tableau à une dimension. En effet, il faut

- allouer la zone mémoire correspondant au pointeur `tab` (tableau de pointeurs, taille = nombre de lignes du tableau)
- allouer les zones mémoire correspondant aux pointeurs `tab[i]` (tableau de nombres, taille = nombre de colonnes du tableau).

Pour libérer la mémoire, il faut procéder en sens inverse :

- libérer les pointeurs `tab[i]`
- libérer le pointeur `tab`

Voici un exemple d'utilisation de ces différentes fonctions

```
int nb_lin = 10;
int nb_col = 20;
double **tab2;        /* Pointeur de pointeur */
/* Allocation */
tab2 = malloc(nb_lin*sizeof(double*)); // Allocation de tab
for(i=0;i<nb_lin;i++) tab2[i] = malloc(nb_col*sizeof(double)); // Allocation de tab[i]
/*Utilisation */
tab2[7][5] = 4;
*(*(tab2+7)+5) = 4; /* ces deux lignes sont équivalentes ! */
/* Libération */
for (i=0;i<nb_lin;i++) free(tab2[i]);
free(tab2);          /* Il faut libérer d'abord les tab2[i], et ensuite tab2 /*
                    /* l'inverse ne marche pas ! */
```

En général, on créera des fonction permettant de réaliser l'allocation et la libération d'un tableau 2D. En généralisant ce principe, on peut facilement créer des tableaux de dimensions 3,4, ... alloués dynamiquement.

7.9 Pointeurs sur des fonctions

Dans certains cas, il est utile de pouvoir passer une fonction en argument d'une autre fonction. C'est par exemple le cas lorsqu'on fait un programme de détermination de racines d'une équation. Il est intéressant de pouvoir entrer la fonction dont on va déterminer la racine en argument de la fonction. De même, dans les fonctions de tri, il peut être utile de changer la fonction de comparaison entre éléments afin de changer les critères de tri.

En C, la transmission d'un fonction en argument d'une autre fonction (on parle de "fonction formelle") se fait en utilisant un "pointeur de fonction". Voici un exemple d'utilisation dans un programme de recherche de racines par dichotomie :

```
double dicho(double a, double b, double eps, double (*f) (double))
{
    double x;
    // On se place dans le cas où f(a)<0
    if ((*f)(a) > 0) {x=a;a=b;b=x;}
    // Début de la boucle
    while (fabs(b-a)>eps)
    {
        x=(a+b)/2;
        if ((*f)(x)<0)
            a=x;
        else
            b=x;
    }
    return x;
}
```

`f` est un pointeur vers une fonction. Donc `*f` désigne une fonction et `(*f)(x)` un appel à cette fonction. En fait, la norme du C a introduit une simplification, qui fait que la notation `f(x)` est équivalente à `(*f)(x)`. Dans la fonction ci-dessus, on peut donc remplacer la ligne

```
if ((*f)(a) > 0) {x=a;a=b;b=x;}
```

par simplement

```
if (f(a) > 0) {x=a;a=b;b=x;}
```

La fonction `dicho` peut être appelée avec une fonction déjà définie, ou avec une fonction définie par l'utilisateur. Par exemple, la fonction `cos` est déjà définie dans `<math.h>` :

```
#include <math.h>
...
x = dicho(0.0,1.0,1E-8,cos);
```

Autre exemple d'utilisation, avec une fonction créée par l'utilisateur :

```
double fonc(double x) { return (x-0.5); }
...
x = dicho(0.0,1.0,1E-8,fonc);
```

Chapitre 8

Les fichiers

Un programme a souvent besoin
- de lire des données dans un fichier
- de sauvegarder des résultats dans un fichier.

Le langage C fournit des fonctions permettant de lire et d'écrire dans des fichiers. Dans la suite de ce chapitre, on décrit les fonctions les plus utiles.

8.1 Ouvrir un fichier : `FILE *fopen (char *nom, char *mode)`

Cette fonction ouvre un fichier dont le nom est indiqué dans la chaîne de caractères `nom`. Elle renvoie un pointeur sur le flot de données [1] correspondant, qui est une structure de type FILE. Ce type est défini dans le fichier `<stdio.h>`, qui doit donc ête mis en en-tête de tout programme utilisant des fichiers. Ce pointeur doit être déclaré au début du programme sous la forme suivante :

```
FILE *fichier
```

Si l'opération échoue (fichier absent, etc.), elle renvoie le pointeur `NULL`.

La fonction `fopen` a deux arguments :
- `nom` est une chaîne de caractères indiquant le nom du fichier, ou bien le nom du flot de données standard (`stdin`, `stdout`, ...) qu'on souhaite utiliser.
- `mode` désigne le type de traitement du flot de donnéses. Les valeurs permises pour ce paramètre sont :
  - `"r"` (read) : ouverture du fichier. Le fichier doit exister, son contenu n'est pas détruit. On commence la lecture au début du fichier. Seule les opérations de lecture sont autorisées (on ne peut pas écrire dans le fichier).
  - `"r+"` : comme `"r"`, mais les opérations d'écriture sont aussi permises.
  - `"w"` (write) : création d'un fichier. Le fichier peut exister ou non, son contenu est totalement détruit. Seules les opérations d'écriture sont permises.
  - `"w+"` : comme `"w"`, mais les opérations de lecture sont aussi permises.

[1] En langage C, un fichier est considéré comme un "flot de donnés" (dataflow), c'est à dire une suite d'octet. On dit donc que la fonction `fopen` sert à "ouvrir" un flot de données, ce qui revient à créer une structure de type `FILE` sur laquelle va pointer le pointeur renvoyé par la fonction. Un flot de données n'est pas forcément un fichier. Par exemple, le flot de données `stdin` correspond aux données tapées au clavier (il ne peut bien évidemment être utilisé qu'en lecture). Le flot de données `stdout` correspond aux données affichées à l'écran (il ne peut bien évidemment être utilisé qu'en lecture).

- `"a"` (append) : allongement d'un fichier. Le fichier existe ou non ; s'il existe, son contenu n'est pas effacé. Le descripteur du flot est positionné en écriture à la fin du fichier. Seules les opérations de d'écriture sont permises.
- `"a+"` : comme `"a"`, mais les opérations de lecture sont aussi permises.

8.2 Fermer un fichier : `int fclose (FILE *flot)`

Cette fonction ferme le flot de données `flot`. Elle produit en particulier l'écriture physique des tampons. Elle renvoie la valeur zéro en cas de succès, et une autre valeur en cas d'erreur.

Oublier de fermer un fichier ouvert en lecture n'a en général pas de conséquences graves. En revanche, oublier de fermer un fichier ouvert en écriture peut avoir des conséquences fatales, il risque de ne pas être écrit correctement sur le disque après l'arrêt du programme.

8.3 Détecter la fin d'un fichier : `int feof (FILE *flot)`

Cette fonction est uniquement utilisée en lecture. Lorsqu'il parcourt un flot de données, le processeur maintient un "pointeur" qui lui indique à tout moment à quel emplacement du flot il se trouve. Grâce à cela, il peut savoir si la fin du fichier a été atteinte. La fonction `feof` renvoie une valeur non nulle si et seulement si la dernière valeur du fichier a été lue. Elle permet donc de détecter qu'on est arrivé à la fin du fichier.

8.4 Exemples d'utilisation

Ci-dessous, on trouvera quelques exemples d'utilisation des fonctions de lecture et d'écriture de fichiers.

8.4.1 Ecriture d'un fichier

```
#include <stdio.h>
void main()
{
    double a,b;
    FILE   *fichier;

    a=1.5;b=2.5;
    /* Ouverture du fichier en ecriture */
    fichier = fopen("C:/goudail/fichier.txt","w");
    /* Verifier que le fichier a bien été ouvert */
    if (fichier != NULL)
    {
        /* Ecriture */
        fprintf(fichier,"%f\n",a);
        fprintf(fichier,"%f\n",b);
        /* Fermeture du fichier */
        fclose(fichier);
    }
}
```

Après exécution de ce programme, le contenu du fichier est :

```
1.5
2.5
```

La syntaxe de `fprintf` est la même que celle de `printf`, excepté le fait qu'on doit spécifier le flot de données en sortie.

En fait, le flot de données associé à une sortie à l'écran est prédéfini dans `<stdio.h>`. Il s'agit de `stdout` (pour standard output). Donc les commandes `printf(...)` et `fprintf (stdout,...)` sont strictement équivalentes.

8.4.2 Ajout de données dans un fichier

```
#include <stdio.h>
void main()
{
    double c = 3.5;
    FILE   *fichier
    /* Ouverture du fichier en ecriture */
    fichier = fopen("C:/goudail/fichier.txt","a");
    /* Verifier que le fichier a bien été ouvert */
    if (fichier != NULL)
    {
        /* Ecriture */
        fprintf(fichier,"%f\n",c);
        /* Fermeture du fichier */
        fclose(fichier);
    }
}
```

Après exécution de ce programme, le contenu du fichier est :

```
1.5
2.5
3.5
```

8.4.3 Lecture d'un fichier

```
#include <stdio.h>
void main()
{
    int     i;
    double   tab[3]
    double   *p;
    FILE    *fichier
    // Ouverture du fichier en lecture
    fichier = fopen("C:/goudail/fichier.txt","r");
    if (fichier != NULL) // Lecture si le fichier a pu etre ouvert
    {
```

```
        p=tab;  // Initialisation du pointeur p au début du tableau
        while (!feof (fichier))
            fscanf(fichier,"%f\n",p++); // Lecture de tout le fichier
        fclose(fichier);                // Fermeture du fichier
    }
    for(i=0;i<3;i++) fprintf(stdout,"%f\n",tab[i]); // Affichage
}
```

La syntaxe de `fscanf` est la même que celle de `scanf`, excepté le fait qu'on doit spécifier le flot données en entrée.

En fait, le flot de données associé à une entrée au clavier est prédéfini dans `<stdio.h>`. Il s'agit de `stdin` (pour standard output). Donc les commandes `scanf(...)` et `fscanf (stdin,...)` sont strictement équivalentes.

8.4.4 Fichiers au format binaire

Dans les exemples précédents, les données écrites et lues dans les fichiers le sont sous forme de caractères. Le fichier contient donc une suite de caractères codés en ASCII. Ce format de codage est adapté aux stockage de chaînes de caractères. Il est également pratique pour stocker des nombres, car il permet par exemple de visualiser le contenu du fichier avec un éditeur de texte. Cependant, dans le cas du stockage des nombres, il n'est pas optimal en terme de volume. Pour réduire l'espace disque occupé par un fichier, il est préférable de stocker les nombres en format binaire, c'est en dire dans le format de données utilisé par la machine.

Soit par exemple le nombre : `1.34564335545454E-18`. Au format `double`, il occupe 8 octets. Au format ASCII, chaque caractère occupe 1 octet, donc il occupe au total 20 octets.

Les instruction de d'écriture et de lecture au format binaire sont les suivantes :

`size_t fwrite(const void  *source,size_t taille, size_t nombre, FILE *flot)`
Cette fonction écrit `nombre` objets de la `taille` indiquée en paramètre, qui se trouvent les uns derrière les autres à partir de l'adresse `source`. La fonction renvoie le nombre d'objets effectivement écrits, qui peut être inférieur au nombre demandé en cas d'erreur.

`size_t fread(void  *destination,size_t taille, size_t nombre, FILE *flot)`
Cette fonction essaye de lire sur le `flot` indiqué `nombre` objets dont la `taille` est indiquée en paramètre. Il copie ces objets à l'adresse pointée par `destination`. La fonction renvoie le nombre d'objets effectivement lu, qui peut être inférieur à `nombre` en raison de la fin du fichier, d'un erreur, etc.

Voici un exemple très simple d'utilisation

```
#include <stdio.h>
#include <math.h>
#define DIM 10000
void main()
{
    int     i;
    double  sum,tab1[DIM],tab2[DIM];
    FILE   *fichier;
```

```
    /* Remplissage du tableau*/
    for(i=0;i<DIM;i++) {tab1[i]=i*atan(1);}
    /* Ecriture au format binaire */
    fichier = fopen("C:/goudail/fichier.bin","wb");
    if (fichier != NULL)
    {
        fwrite(tab1,sizeof(double),DIM,fichier);
        fclose(fichier);
    }
    // Lecture du fichier
    fichier = fopen("C:/goudail/fichier.bin","rb");
    if (fichier != NULL)
    {
        fread(tab2,sizeof(double),DIM,fichier);
        fclose(fichier);
    }
    /* Affichage*/
    sum=0;
    for(i=0;i<100;i++) {sum=sum+tab1[i]-tab2[i];}
    printf("Difference : %f\n",sum);
}
```

Chapitre 9

Les chaînes de caractères

Dans un programme en C, on a très souvent l'occasion de manipuler des chaînes de caractères. Les chaînes de caractères peuvent être considérées comme des tableaux de type `char`, avec quelques propriétés particulières. Par conséquent, il existe dans la bibliothèque standard du langage C des fonctions permettant de les manipuler facilement.

9.1 Déclaration et allocation d'un chaîne de caractères

En langage C, une chaîne de caractères est un tableau de caractères (type `char`) dont le dernier élément est le caractère nul noté `"\0"` (caractère de terminaison). Comme pour tous les tableaux, on doit leur allouer un espace mémoire. Cela peut se faire de manière statique ou dynamique.

```
char   chaine1[100];           // allocation statique
char   *chaine2;
...
chaine2 = malloc (100*sizeof(char));    // allocation dynamique
```

ATTENTION : Pour définir une chaine de caractères, **il ne suffit pas de la déclarer comme un pointeur** sur un caractère (`char *chaine2;`) : il faut aussi lui allouer un espace mémoire avec `malloc`.

9.2 Donner un contenu à une chaîne de caractères

Attention !Pour donner un contenu à une chaîne de caractères, on ne peut pas utiliser l'opérateur `=`, sauf lors de la déclaration de la chaîne (donc, dans le cas d'une allocation statique).

```
char    chaine1[100]="eso";  /* OK */
char    chaine2[100]="ose";  /* OK */
chaine1 = "ose";              /* Interdit ! */
chaine1 = chaine2;            /* Interdit ! */
```

La dernière commande ne recopie pas `chaine2` dans `chaine1`. Elle fait pointer le pointeur `chaine1` sur l'emplacement mémoire pointé par `chaine2`. La zone mémoire pointée initialement par `chaine1` est perdue, car on a perdu le pointeur qui permettait de la localiser.

Pour recopier le contenu de `chaine2` dans l'emplacement pointé par `chaine1`, il faut utiliser la fonction `strcpy`, qui est contenue dans la librairie standard `<string.h>`.

```
#include <string.h>
char    chaine1[100]="eso";
char    chaine2[100]="ose";
strcpy(chaine1,chaine2);   /* OK */
printf("%s\n%s\n",chaine1,chaine2);
```

Remarque : le caractère de terminaison est automatiquement ajouté par la fonction `strcpy`. Il est inutile de l'indiquer dans la chaîne (`"eso\0"` est inutile, `"eso"` suffit).

Le nombre maximal de caractères qu'une chaine peut contenir est défini au moment de la déclaration (allocation statique) ou de l'allocation dynamique. En revanche, le nombre réel de caractère de la chaîne, c'est à dire, le nombre de caractères se trouvant avant le caractère de terminaison `\0`, peut être obtenu à l'aide de la fonction `strlen` définie dans `<string.h>` :

```
char    chaine1[100]="clou";
char    chaine2[100]="marteau";
printf("%d\n",strlen(chaine1));   /* affiche 4 */
printf("%d\n",strlen(chaine2));   /* affiche 7 */
```

9.3 Autres fonctions de base

La bibliothèque `<string.h>` contient de nombreuses fonctions de manipulations de chaînes de caractères. En voici quelques unes.

• `char *strcpy(char *destination,const char *source)` [1]
Copie la chaîne de caractères `source` à l'adresse `destination`. Elle renvoie `destination`.

• `char *strcat(char *destination,const char *source)`
Copie la chaîne de caractères `source` à la suite de la chaîne `destination`. Elle renvoie `destination`.

• `char *strcmp(const char *destination,const char *source)`
Compare les chaînes `a` et `b` en fonction de l'ordre lexicographique. Elle renvoie une valeur négative, nulle ou positive selon que `a` est inférieur, égal ou supérieur à `b`.

• `int strlen(const char *chaine)`
Renvoie le nombre de caractères utiles de `chaine` (c'est à dire sans compter le caractère de terminaison `\0`).

• `char *strrev(char *chaine)`
Inverse l'ordre des caractères contenus dans la chaîne caractères `chaine`. La fonction renvoie le pointeur sur la chaine modifiée (c'est à dire sur la chaîne d'entrée `chaine`).

9.4 Construire une chaîne de caratères : la fonction `sprintf`

Il est également utile de pouvoir construire des chaînes de caractères selon un format déterminé. Cela peut se faire avec la fonction suivante, définie dans `<stdio.h>` :

• `int sprintf (char *destination, const char *format)`

[1] Dans le deuxième argument de cette fonction, le qualifieur `const` sert à spécifier que le contenu des cases mémoires pointées par le pointeur `*source` ne peut pas être modifié dans la fonction.

Elle effectue exactement la même mise en forme que `printf`, en particulier, elle utilise la même syntaxe pour définir le format (argument `format`). La seule différence, c'est qu'elle stocke le résultat dans la chaîne de caractères `*destination` mise en argument.

Voici un exemple d'utilisation :

```
char    chaine[100], ext[4]="tif";
int     i=4,val=5;

// Construction d'une chaine de caracteres
sprintf(chaine,"file_%d_par_%d.%s",i,val,ext);
printf("%s\n",chaine); // affiche "file_4_par_5.tif"
```

On voit que la fonction `sprintf` permet de construire une chaîne de caractères selon un certain format, en fonction de différents paramètres (ici, `i,val,ext`) qui peuvent être de types différents.

9.5 Analyse d'une chaîne de caractères : la fonction `sscanf`

La fonction

- `int sscanf (char *source, const char *format)`

réalise la même analyse lexicale que `scanf`, mais à partir d'une chaine de caractères `source` entrée en argument de la fonction.

Voici un exemple d'utilisation :

```
char    chaine[100]="file_4_par_5.tif";
char    ext[4];
int     i,val;

// Analyse lexicale d'une chaine de caractères
sscanf(chaine,"file_%d_par_%d.%s",&i,&val,ext);
printf("%d,%d,%s\n",i,val,ext); // affiche "4,5,tif"
```

La fonction `sscanf` permet d'analyser la chaîne selon un certain format, et de la décomposer en élément plus simples, qui sont mis dans les variables `i,val,ext`.

Attention ! Il y a un problème particulier lorsqu'on veut séparer une chaîne de caractères en plusieurs sous-chaînes. Pour la fonction `sscanf`, une chaîne de caractères est terminée par un espace blanc ou le caractère nul de fin de chaîne. Ainsi, les instructions

```
strcpy(chaine,"tic tac");
sscanf(chaine,"%s %s",chaine1,chaine2);
printf("%s\n",chaine1); printf("%s\n",chaine2);
```

affichent

```
tic
tac
```

En revanche,

```
strcpy(chaine,"tic_tac");
sscanf(chaine,"%s_%s",chaine1,chaine2);
printf("%s\n",chaine1); printf("%s\n",chaine2);
```

affiche

```
tic_tac
```

La variable `chaine2` est vide. Pour résoudre ce problème, il faut spécifier les caractères que peut contenir la chaîne qu'on souhaite lire. Pour cela, on remplace le suffixe `s` avant le `%` par la liste des caractères que peut contenir la chaîne qu'on souhaite lire :

```
%[liste des caracteres possibles]
```

Si on souhaite au contraire spécifier les caractères que ne doit pas contenir la chaîne, on précède la liste de ces caractères par le symbole `^` :

```
%[^liste des caracteres non souhaités]
```

Les listes de caractères sont simplement des chaînes de caractères énumérants les caractères possibles. Ainsi, les instructions suivantes

```
strcpy(chaine,"tic_tac");
sscanf(chaine,"%[abcdefghijklmnopqrst]_%s",chaine1,chaine2);
printf("%s\n",chaine1); printf("%s\n",chaine2);
```

affichent le résultat voulu :

```
tic
tac
```

En effet, la lecture du premier paramètre `chaine1` s'arrête lorsqu'on rencontre le caractère `'_'`, qui n'est pas la liste `abcdefghijklmnopqrst`. Le même résultat est obtenu avec

```
strcpy(chaine,"tic_tac");
sscanf(chaine,"%[a-t]_%s",chaine1,chaine2);
printf("%s\n",chaine1); printf("%s\n",chaine2);
```

car la syntaxe `[a-t]` signifie "tous les caractères compris entre `a` et `t`". On obtient également le même résultat avec :

```
strcpy(chaine,"tic_tac");
sscanf(chaine,"%[^_]_%s",chaine1,chaine2);
printf("%s\n",chaine1); printf("%s\n",chaine2);
```

En effet, la présence du caractère `'^'` au début de la liste signifie que le caractère `'_'` ne doit pas apparaître dans la chaîne lue. La lecture de la première chaîne `chaine1` s'arrête lorsqu'on rencontre le caractère `'_'`.

Chapitre 10

Les structures

Jusqu'à maintenant, nous avons utilisé des variables sclaires (nombres, caractères, ...) et des tableaux, qui sont des ensembles de variables de même type.

En C, on peut également définir des variables "composites", constituées d'un ensemble de variables de types différents. Ces variables portent le nom de *structures*.

10.1 Structure

La structure suivante permet de définir un nombre complexe à partir de sa partie imaginaire et de sa partie réelle :

```
struct  complexe    /* Definition de la structure */
{
    double  re;     /* Partie réelle */
    double  im;     /* Partie imaginaire */
};  // ne pas oublier le point-virgule !

struct complexe z;  /* Déclaration d'un objet de ce type */

z.re = 1; z.im =2;  /* Intialisation des champs */
```

Le mot `struct` est un mot réservé du langage C. Il permet de définir une structure, qui est un nouveau type de données, au même titre que `double`, `int`, etc. Ce type de donnée s'appelle `struct complexe`. La variable `z` est une variable du type `struct complexe`.

Cette structure comprend deux champs, qui sont des variables de type `double`. La syntaxe permettant d'accéder champs d'une structures est indiquée dans l'exemple.

On peut simplifier l'écriture en utilisant l'instruction `typedef`, qui permet de donner un nom à ce nouveau type de données. Attention, la structure est alors définie dans l'en-tête du programme, pas dans le `main` :

```
struct  complexe    /* Definition de la structure */
{
    double  re;     /* Partie réelle */
    double  im;     /* Partie imaginaire */
}
```

```
/* Définition du type de données COMPLEXE à partir */
/* de la structure "struct complexe" */
typedef struct complexe COMPLEXE;
...
void main
{
    COMPLEXE z       /* Déclaration de la variable z de type COMPLEXE */
    ...
}
```

Par tradition, les nouveaux types de données définis par `typedef` sont notés en majuscules. On peut regrouper la définition de la structure et du type de la manière suivante, qui est équivalente à l'écriture précédente :

```
typedef struct  complexe { double  re; double  im; } COMPLEXE;
```

Le nouveau type de données ainsi défini peut être utilisé comme les autres. Par exemple, il peut être utilisé comme argument d'une fonction. La fonction suivante calcule la somme de deux nombres complexes.

```
COMPLEXE somme(COMPLEXE z1,COMPLEXE z2)
{
    COMPLEXE    z;
    z.re = z1.re + z2.re;
    z.im = z1.im + z2.im;
    return z;
}
```

Comme nous l'avons dit, les champs d'une structure peuvent ne pas avoir le même type :

```
typedef struct client
{
    char    nom[50];    /* nom du client */
    char    prenom[50]; /* prénom du client */
    double  montant;    /* somme disponible sur le compte */
} CLIENT;
```

On peut également construire des structures à partir de structures déjà existantes. Dans l'exemple suivant, la structure définit le vecteur de Jones de la lumière, qui est un vecteur complexe à deux dimensions.

```
typedef struct jones {COMPLEXE jx;COMPLEXE jy;} JONES;
```

Les structures s'utilisent comme les autres types de variables. On peut définir des pointeurs sur des structures, et donc des tableaux de structures. La fonctions suivante permet d'initialiser à zéro un nombre complexe.

```
#include <stdlib.h>
...
void initialise(COMPLEXE *z)
{
    z->re = 0; z->im = 0;
}
```

Il faut noter que l'argument de cette fonction est un pointeur sur une structure, puisque le contenu de la structure doit être modifié dans la fonction.

La syntaxe `z->re` désigne le contenu du champ `re` de la structure pointée par `z`. C'est un raccourci pour `(*z).re`.

La fonctions suivante permet d'initialiser à zéro un tableau de nombres complexes

```
void initialise_tableau(COMPLEXE *z,int dim)
{
    int i;
    for (i=0;i<dim;i++) initialise(&(z[i]));
}

main()
{
    COMPLEXE * tab;
    int      i,dim=10;
    /* allocation */
    tab =malloc(dim*sizeof(COMPLEXE));
    /* intialisation */
    initialise_tableau(tab,dim);
    // Affichage
    for(i=0;i<dim;i++) printf("%f\t%f\n",tab[i].re,tab[i].im);
}
```

On peut noter que l'opérateur `sizeof` permet de déterminer la taille du type de données `COMPLEXE`.

ATTENTION La notion de stucture est utile, mais il faut l'utiliser avec modération. Il faut limiter son utilisation aux cas où elle est indispensable, comme les listes chaînées par exemple. En effet, leur utilisation peut ralentir considérablement les programmes.

10.2 Listes chaînées

Les tableaux sont des outils très utiles car ils permettent de manipuler un grand nombre de données à la fois. L'inconvénient, c'est qu'il ont une taille fixe. De fait, il est difficile de réduire leur taille, ou l'augmenter de manière dynamique. Pour résoudre ce problème, on utilise les listes chaînées. Ce sont des structures de données qui ne sont pas définies dans le C standard, il faut donc les définir, ainsi que les fonction qui les manipulent. Pour cela, on va utiliser des structures.

Une liste chaînée est un ensemble d'éléments, appelés "nœuds", où chaque nœud "pointe" sur le nœud suivant. Les avantages d'une telle structure sont :

- On peut ajouter un nœud à la fin de la liste, donc augmenter sa taille progressivement.
- On peut aussi insérer un nœud entre deux nœuds déjà existant (ce qu'on ne pouvait pas faire avec un tableau)
- On peut supprimer un nœud, ce qui permet de diminuer progressivement la taille de la liste

La contrepartie de ces avantages sera qu'on ne pourra pas accéder à un élément donné aussi facilement que dans un tableau, où il suffit de connaître l'indice de l'élément dans le tableau. Il nous faudra parcourir la chaîne nœud par nœud.

10.2.1 Structure d'un nœud

Un nœud de la liste chaînée doit contenir :
- Des données
- Un pointeur sur le nœud suivant

Il contient donc des données de types différents, et doit être défini sous forme de structure. Voici par exemple une structure où la donnée représentée par le nœud est simplement un caractère.

```
typedef struct noeud {char val;struct noeud *suivant;} NOEUD;
```

Notez que cette structure est définie de manière *récursive*. En effet, l'un de ses éléments est un pointeur sur la structure de type `NOEUD` qu'on est justement en train de définir. Le langage C permet cela, sans quoi il serait impossible de définir des listes chaînées de cette manière.

10.2.2 Création d'une liste chaînée

La première étape est la création d'un liste chaînée. Cela revient à créer un pointeur `debut`, dont le champ `suivant` est égal au pointeur `NULL`. Ce premier nœud n'est pas vraiment un élément de la liste. Il matérialise simplement son point de départ. C'est pourquoi nous fixerons arbitrairement la valeur de son champ `val`. On aboutit à la fonction suivante :

```
NOEUD* LC_Init ()
{
    NOEUD* debut;

    // Allocation et initialisation de debut
    debut = malloc(sizeof(NOEUD));
    debut->val='0';
    debut->suivant=NULL;

    return debut;
}
```

On notera que la fonction retourne le pointeur sur le début de la liste.

10.2.3 Insertion d'un élément au milieu d'une liste chaînée

Pour ajouter des éléments à la chaîne, il faut disposer d'une fonction permettant d'insérer un nouveau nœud, contenant une donnée, après un autre nœud. Cela peut se faire avec la fonction suivante :

```
NOEUD* LC_Insere (char c, NOEUD* n_courant)
{
    NOEUD* n_ajoute;

    // Allocation et initialisation du noeud à ajouter
    n_ajoute = malloc(sizeof(NOEUD));
    n_ajoute->val = c;
    // "Raccordement" du noeud ajouté
    n_ajoute->suivant =n_courant->suivant;
```

```
    n_courant->suivant = n_ajoute;

    return(n_ajoute);
}
```

On notera que la fonction renvoie un pointeur sur le nœud qui a été ajouté.

10.2.4 Autres fonctions

Parmi les autres fonctions utiles pour gérer les listes chaînées, on peut citer :

- une fonction qui compte le nombre de nœuds d'une liste chaînée.
- une fonction qui affiche les valeurs de tous les nœuds situé après noeud d'indice n.
- une fonction de suppression d'un nœud situé après le noeud d'indice n.

Chapitre 11

Bibliographie

Le présent document résume les notions développée dans le cours sur les notions de base du langage C. C'est une simple introduction, et non un ouvrage de référence. Voici une liste d'ouvrages plus exhaustifs :

- Brian W. Kernighan, Dennis M.Ritchie, *Le Langage C*, Manuels informatiques Masson.
- Claude Delannoy, *Langage C*, Eyrolles.

On peut aussi trouver des polys plus complets sur Internet. Par exemple,

- `www.dil.univ-mrs.fr/ garreta/Polys/PolyC.pdf`
- http ://www-ipst.u-strasbg.fr/pat/program/tpc.htm

Annexe A

Annexes

A.1 Mots-clés

Les mots suivants sont réservés. Leur fonction est prévue par la syntaxe du langage C et ils ne peuvent être utilisés dans un autre but :

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

A.2 Code ASCII

Voir page suivante.

Code ASCII

Code	*Caractère*	*Code*	*Caractère*	*Code*	*Caractère*	*Code*	*Caractère*	*Code*	*Caractère*
0	[car. nul]	69	E	116	t	164	¤	211	Ó
…		70	F	117	u	165	¥	212	Ô
7	[sig. sonore]	71	G	118	v	166	¦	213	Õ
8	[ret. arrière]	72	H	119	w	167	§	214	Ö
9	[tabulation]	73	I	120	x	168	¨	215	×
10	[saut ligne]	74	J	121	y	169	©	216	Ø
11	[tab. vert.]	75	K	122	z	170	ª	217	Ù
12	[saut page]	76	L	123	{	171	«	218	Ú
13	[ret. chariot]	77	M	124	|	172	¬	219	Û
...		78	N	125	}	173	-	220	Ü
32	[espace]	79	O	126	~	174	®	221	Ý
33	!	80	P	…		175	¯	222	Þ
34	"	81	Q	128	€	176	°	223	ß
35	#	82	R	…		177	±	224	à
36	$	83	S	130	‚	178	²	225	á
37	%	84	T	131	*ƒ*	179	³	226	â
38	&	85	U	132	„	180	´	227	ã
39	’	86	V	133	…	181	µ	228	ä
40	(	87	W	134	†	182	¶	229	å
41	)	88	X	135	‡	183	·	230	æ
42	*	89	Y	136	ˆ	184	¸	231	ç
43	+	90	Z	137	‰	185	¹	232	è
44	,	91	[	138	Š	186	º	233	é
45	-	92	\	139	‹	187	»	234	ê
46	.	93	]	140	Œ	188	¼	235	ë
47	/	94	^	…		189	½	236	ì
48	0	95	_	142	Ž	190	¾	237	í
49	1	96	`	…		191	¿	238	î
50	2	97	a	145	‘	192	À	239	ï
51	3	98	b	146	’	193	Á	240	ð
52	4	99	c	147	“	194	Â	241	ñ
53	5	100	d	148	”	195	Ã	242	ò
54	6	101	e	149	•	196	Ä	243	ó
55	7	102	f	150	–	197	Å	244	ô
56	8	103	g	151	—	198	Æ	245	õ
57	9	104	h	152	˜	199	Ç	246	ö
58	:	105	i	153	™	200	È	247	÷
59	;	106	j	154	š	201	É	248	ø
60	<	107	k	155	›	202	Ê	249	ù
61	=	108	l	156	œ	203	Ë	250	ú
62	>	109	m	…		204	Ì	251	û
63	?	110	n	158	ž	205	Í	252	ü
64	@	111	o	159	Ÿ	206	Î	253	ý
65	A	112	p	160	[espace]	207	Ï	254	þ
66	B	113	q	161	¡	208	Ð	255	ÿ
67	C	114	r	162	¢	209	Ñ		
68	D	115	s	163	£	210	Ò		

Les codes non cités ne sont pas gérés par Microsoft Windows.