UPMC
SORBONNE UNIVERSITÉS

Programmation et structures de données en C
UE 2I001. Poly de cours.

Auteur: S. Doncieux
Date: 1er septembre 2015

TABLE DES MATIÈRES

1 INTRODUCTION

Ce premier chapitre présente le contenu de l'unité d'enseignement (UE) « Programmation et structures de données en C » (2I001). Cette UE n'a pas pour but d'introduire le langage C, mais d'approfondir les connaissances acquises précédemment. Les notions de base seront rappelées rapidement avant de passer aux notions importantes présentées au cours de cette UE. Ce support de cours s'appuie sur les cours donnés dans l'UE LI215 puis 2I001 depuis 2005. Ces cours ont été conçus conjointement par Jean-Lou Desbarbieux et Stéphane Doncieux.

1.1 OBJECTIFS PÉDAGOGIQUES

Le module 2I001 a pour objectif d'approfondir les connaissances en C d'étudiants ayant suivi l'initiation de l'UE 1I002. Là où l'UE 1I002 se focalise sur les bases de la programmation impérative en C en simplifiant son utilisation, l'UE 2I001 présente le C standard, sans masquer quoi que ce soit. Les fonctions utilisées font partie de la libC, c'est-à-dire de la bibliothèque standard associée au langage C.

Au cours de cette UE, nous verrons comment manipuler la mémoire avec toutes les conséquences que cela peut avoir en C : notion de *pointeurs*, *passage d'arguments*, *tableaux*, *allocation dynamique de mémoire*, etc. Nous aborderons également la notion de *structures*, notamment au travers des *listes chaînées* et des *arbres*, structures très fréquemment utilisées en informatique et pour lesquelles la manipulation de pointeurs est essentielle lorsqu'elles sont implantées en C. Nous évoquerons rapidement l'intérêt de chacune de ces structures. Cependant, leur utilisation n'a pour but que d'illustrer l'utilisation de structures et la manipulation de la mémoire sur des exemples classiques en informatique. **Il n'est pas du ressort de ce cours** que de présenter en détail les avantages et inconvénients de ces structures ainsi que les différents algorithmes les manipulant avec leur efficacité, ces différents points seront cependant évoqués rapidement dans les exercices.

Nous présenterons les principes de la compilation et nous utiliserons le compilateur open-source `gcc` [1] disponible sur la plupart des systèmes d'exploitation (linux, MacOSX, Microsoft Windows). De nombreux environnements de développement existent qui permettent d'inclure un éditeur de texte avec des boutons permettant de créer directement l'exécutable associé,

1. http://gcc.gnu.org

voire de déboguer le programme, que ce soit `Geany` ou encore `Code::Blocks`, pour ne citer que des environnements open source. Si l'usage de tels environnements facilite souvent le développement, il cache parfois les opérations qu'impliquent la programmation et la compilation d'un programme.

Lien entre polycopié de cours, transparents et polycopiés de TD/TP

Le présent polycopié de cours rassemble l'ensemble des connaissances théoriques à acquérir au cours de l'UE, organisées de manière thématique. L'objectif est de pouvoir s'y référer à chaque instant de la formation pour y trouver les informations qui manquent sur un point spécifique. Mais **les supports principaux pour la progression dans cette UE sont les transparents de cours et le polycopié de TD/TP**. Ils sont structurés en « semaines d'enseignement », chacune d'elles abordant une notion spécifique du langage nécessitant la maîtrise des précédentes.

Chaque exercice est assorti d'un niveau de difficulté :

— *base* désigne un exercice d'application directe du cours. Vous devriez systématiquement faire tous les exercices de base.

— *obligatoire* désigne un exercice qu'il est indispensable de faire pour poursuivre la progression.

— *entrainement* désigne un exercice de niveau intermédiaire, que tous les étudiants devraient être capables de faire à la fin d'une semaine d'enseignement. Les exercices donnés lors de l'examen seront du même niveau. Un étudiant capable de faire tous les exercices d'entrainement ne devrait pas avoir de difficulté à l'examen.

— *approfondissement* désigne un exercice difficile destiné aux étudiants qui souhaiteraient aller plus loin. Ces exercices sont plus difficiles que ce que nous vous demanderons à l'examen, mais ils sont parfaitement faisables avec les seules connaissances de l'UE 2I001.

1.5 OUVRAGES DE RÉFÉRENCE

Le C est un langage très courant, de nombreux ouvrages lui sont consacrés. Nous en avons listé ici quelques uns.

Programmer en langage C, par Claude Delannoy, chez Eyrolles est un ouvrage accessible et très clair.

Le langage C : norme ANSI, par Brian W. Kernighan & Denis M. Ritchie, chez Dunod. Appelé familièrement « le Kernighan & Ritchie », est LA référence du langage.

Langage C par Samuel P. Harbison & Guy L. Steele, chez Dunod, est à la fois un bon ouvrage de référence et une bonne description des principales

bibliothèques disponibles en standard. Il vous permettra de faire une bonne transition entre 1I002 et 2I001.

Certains ouvrages de qualité ne sont malheureusement pas disponibles en français, c'est le cas notamment de *C : a software engineering approach*, par Peter A. Darnell & Philip E. Margolis, chez Springer-Verlag, qui présente une très bonne réflexion sur l'écriture de programmes d'envergure en C. De même, *C programming : a modern approach* par K. N. King publié chez Norton & Company, est reconnu comme un ouvrage de référence pour la clarté et l'étendue de son contenu, auprès des programmeurs tant débutants que confirmés.

2 RAPPELS ET NOTIONS DE BASE

Ce chapitre contient un rappel des principales notions sur lesquelles nous nous appuierons dans la suite de cette UE. Il s'agit bien ici d'un rappel de notions qui ont été vues au cours de l'UE 1I002. Au cas où les points évoqués ici ne seraient pas maîtrisés, nous vous conseillons de revoir le cours de 1I002 ou de consulter un des ouvrages mentionnés. Certaines notions, nécessaires pour les premiers TD/TP, seront rappelées ici brièvement avant d'être reprises plus en détails ultérieurement.

2.1 BREF HISTORIQUE

Le langage C a été inventé en 1972 par Dennis Ritchie et Ken Thompson (AT&T Bell Laboratories) pour réécrire Unix et développer des programmes sous Unix.

En 1978, Brian Kernighan et Dennis Ritchie publient la définition classique du C dans le livre The C Programming language.

C est une norme ANSI (ANSI-C) depuis 1989 et un standard ISO depuis 1990. Cela signifie que toutes les spécificités du langage ont été discutées et sont détaillées dans le descriptif de la norme. Ainsi le comportement de tout compilateur respectant une de ces normes peut être connu en consultant ces documents. Une nouvelle norme a été proposée en 1999, la norme C99, puis en 2011 la norme C11. Ce cours suit essentiellement la norme de 1989.

2.2 QU'EST-CE QUE LE LANGAGE C ?

Le langage C est un **langage impératif**, autrement dit constitué d'une séquence d'instructions modifiant l'état du programme tel que décrit par des variables.

Le C est un **langage typé**, autrement dit chaque variable a un type. Par contre, il s'agit d'un typage faible dans la mesure où l'on peut manipuler la mémoire sans savoir ce qu'elle contient.

En C, le programmeur doit manipuler directement la mémoire. Il peut allouer la mémoire dont il a besoin et doit ensuite la libérer là où d'autres langages, comme Java simplifient la gestion de la mémoire en prenant en charge automatiquement la libération de mémoire.

2.3 QU'EST-CE QU'UN PROGRAMME, QU'EST-CE QU'UN EXÉCUTABLE ET COMMENT PASSER DE L'UN À L'AUTRE ?

Le C est un **langage compilé** : un programme est décrit dans un fichier de texte, écrit à partir d'un éditeur de texte qui n'a besoin d'aucune fonctionnalité particulière, si ce n'est écrire des fichiers "texte" simples (et non avec un format complexe incluant la mise en page, voire une compression du fichier, comme c'est le cas pour Word ou Open Office).

Le programme doit ensuite être traduit dans un langage compréhensible par le processeur. C'est la tâche du compilateur qui va interpréter le programme tel qu'écrit par le programmeur et le traduire en langage machine. Le résultat de cette étape est un exécutable qui est impossible à comprendre pour un humain (ou tout du moins pas facile du tout !) mais que le processeur peut exécuter. Nous verrons au chapitre 4 que ce processus se fait en plusieurs étapes, mais il n'est pas nécessaire de le savoir à ce stade.

Chaque programme contient une et une seule fonction appelée `main`. C'est elle qui est lancée lorsque l'exécutable est appelé.

Voilà un exemple de programme simple (dans un fichier `welcome.c`), que vous pouvez taper dans votre éditeur de texte préféré. Il est normal que vous ne compreniez pas encore l'utilité de chacune des lignes de ce fichier. Nous allons les passer en revue rapidement, mais nous aurons l'occasion d'y revenir plus longuement par la suite.

```
1 #include<stdio.h>
2
3 int main(void){
4    int reponse=42;
5    printf("Bonjour, bienvenue a l'UE 2I001.\n");
6    printf(" La reponse à toutes vos questions est:");
7    printf(" %d !\n",reponse);
8    return 0;
9 }
```

La fonction `printf` permet d'afficher des chaînes de caractères, mais aussi des variables. Nous reviendrons plus tard sur cette fonction ; à ce stade, notez qu'elle nécessite au moins un argument de type chaîne de caractères (caractères entourés par des guillemets doubles), appelé format. Si ce premier argument contient des codes de format (caractère pourcent suivi d'un code), d'autres arguments doivent suivre, un par code de format. Ici, il y a un code de format dans le troisième `printf` : %d, il y a donc un autre argument et un seul dans cet appel à cette fonction : `reponse`. Le code suivant le caractère % indique le type de la variable. Les types les plus courants sont les suivants :

— `%d` : entier
— `%c` : caractère
— `%s` : chaîne de caractères
— `%f` : nombre réel

La fonction `printf` nécessite l'ajout de la première ligne :

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

Nous reviendrons plus tard sur ce point.

Avant de voir le résultat de ce programme, il vous faut le compiler en tapant la commande suivante dans un terminal [1] (le répertoire courant du terminal doit être celui qui contient le fichier `welcome.c`) :

```
gcc -o welcome welcome.c
```

Vous pouvez ensuite voir le résultat de l'exécution de ce programme en tapant dans un terminal la commande :

```
./welcome
```

ATTENTION : ne pas oublier le './', nous reviendrons plus tard sur la raison de cela.

IMPORTANT : Pour observer le résultat de l'exécution des instructions d'un programme, il faut réaliser les étapes suivantes (quel que soit l'environnement utilisé) :

1. écrire le programme dans un éditeur de texte
2. compiler le programme
3. lancer l'exécutable créé pendant l'étape de compilation

2.4 TYPES

2.4.1 Types entiers

Les types entiers sont caractérisés par le fait qu'ils représentent des entiers positifs ou nuls seulement (non signés) ou bien positifs ou négatifs (signés). Ils sont également caractérisés par la taille qu'ils occupent en octets, cette taille définissant le nombre de valeurs possibles. Pour certains types, la taille dépend de l'architecture du processeur utilisé (16 bits, 32 bits ou 64 bits).

1. nous reviendrons plus tard sur l'utilisation d'un terminal, cette notion n'est pas essentielle à ce stade.

Type	Signification	Taille (o)	Plage de valeurs
char	Caractère	1	-128 à 127
unsigned char	Caractère	1	0 à 255
short int	Entier court	2	-32768 à 32767
uns. short int	Entier court non s.	2	0 à 65535
int	Entier	2 (16 b) 4 (32 et 64 b)	-32768 à 32767 -2 147 483 648 à 2 147 483 647
unsigned int	Entier non signé	2 (16 b) 4 (32 et 64 b)	0 à 65 535 0 à 4 294 967 295
long int	Entier long	4 8 (64 b)	-2 147 483 648 à 2 147 483 647 -9 223 372 036 854 775 080 à 9 223 372 036 854 775 807
uns. long int	Entier long non s.	4	0 à 4 294 967 295

À noter qu'il existe un type `size_t` qui apparaîtra dans des fonctions de la librairie standard. Il s'agit d'un type d'entier non signé utilisé pour représenter des tailles (notamment des tailles en mémoire). Selon l'implémentation, ce sera équivalent à un `unsigned int` ou à un `unsigned long int`.

2.4.2 Types à virgule flottante

Les nombres réels sont représentés sous la forme suivante : $signe \times mantisse \times base^{exposant}$

Les valeurs données ci-dessous sont celles qui sont utilisées pour un système Linux. La norme C90 est relativement floue sur la représentation des nombres réels [2].

Type	Signification	Taille (o)	Plage de valeurs
float	Simple précision	4	+/- 1.175494e-38 à 3.402823e+38
double	Double précision	8	+/- 2.225074e-308 à 1.797693e+308
long double	Double préc. long	12	+/- 3.362103e-4932 à 1.189731e+4932

ATTENTION : les flottants sont représentés de manière approchée. Une conséquence importante est que l'associativité $(a + (b + c) = (a + b) + c)$ n'est plus garantie : a peut être négligeable devant $b + c$, mais $a + b$ peut ne pas être négligeable devant c. Les deux calculs peuvent donc donner des résultats (très légèrement) différents. Ainsi il n'est pas recommandé de tester des valeurs réelles exactes, mais plutôt de tester que l'écart à la valeur attendue est inférieure à un seuil arbitrairement petit. Par exemple :

2. La norme C99 est plus précise sur ce point.

```
1 float x=0.1,y=0.1;
2 if (fabs(x+y - 0.2)<0.0001)
3     ...
```

doit être préféré à :

```
1 float x=0.1,y=0.1;
2 if (x+y == 0.2) /* peut etre vrai ou faux */
3     ...
```

`fabs` correspondant à la valeur absolue.

2.4.3 Types énumérés

Les types énumérés permettent d'associer des étiquettes à des constantes. Du point de vue du C, cela reste des entiers. Ces étiquettes servent uniquement à améliorer la lisibilité des programmes écrits.

Exemple :

```
1   enum mois {JAN = 1, FEV, MAR, AVR, MAI, JUIN, JUILL,
2      AOUT, SEPT, OCT, NOV, DEC};
3
4   enum mois m=3;
5   enum mois n=JAN;
6   if ( m == MAR ) {
7     printf("m=MAR\n");
8   }
```

La déclaration d'un type énuméré se fait avec le mot-clé `enum` suivi du nom que l'on souhaite donner au type énuméré, puis des étiquettes entourées par des accolades.

Si aucune valeur n'est précisée par un =, la première étiquette se voit attribuée la valeur 0. Les autres valent 1 plus la valeur de l'étiquette précédente.

La déclaration d'une variable de type énuméré se fait ensuite en reprenant le mot-clé `enum` suivi du nom de type choisi et du nom de la variable. La variable ainsi déclarée peut ensuite être initialisée en utilisant les noms d'étiquette ou bien des entiers.

2.5 TABLEAUX

Les tableaux permettent de stocker dans des espaces mémoire contigus des éléments de même type. Leur déclaration se fait en ajoutant des crochets au nom de la variable. La taille du tableau est indiquée entre les crochets. Le

tableau peut être initialisé dès sa déclaration en indiquant les différentes valeurs entre accolades. Si toutes les valeurs du tableau ne sont pas spécifiées, seules les premières cases sont initialisées, les autres contiennent une valeur indéterminée (qui n'est pas nécessairement 0).

Si le tableau est initialisé, il n'est pas nécessaire de spécifier sa taille, le compilateur peut la déterminer à partir du nombre d'éléments donnés pour l'initialisation.

Exemples :

```
1        int T1[3]={2, 1, 5};
2        int T2[]={4, 1, 5};
3        int T2[]; /* impossible! */
4        char TC[3][2]={{1,2},{3,4},{5,6}};
```

2.6 OPÉRATEURS

Les opérateurs existants en C sont les suivants (par ordre de précédence décroissant [3]) :

- référence : () [] -> .
- unaire : ! ~ ++ -- + - * & (type) sizeof
- arithmétique : * / %
- arithmétique : + -
- décalage : << >>
- relationnels : < <= > >=
- relationnels : == !=
- manipulation de bits : &
- manipulation de bits : ^
- manipulation de bits : |
- logique : &&
- logique : ||
- conditionnel : ? :
- affectation : = += -= *= /= %= &= ^= |= <<= >>=

L'opérateur '()' est l'opérateur d'appel d'une fonction et l'opérateur '[]' est utilisé pour accéder à une case d'un tableau.

Les opérateurs '.', '->', '*' (dans sa version unaire) et '&' (dans sa version unaire également) seront présentés ultérieurement, ils concernent les structures et les pointeurs.

Les opérateurs '!', '&&' et '||' sont des opérateurs logiques correspondant respectivement à la négation logique, au ET logique et au OU logique (ou inclusif).

3. L'ordre de précédence indique l'ordre dans lequel ils sont considérés dans le cas d'une expression contenant plusieurs opérateurs.

Les opérateurs arithmétiques permettent de faire des calculs (addition, soustraction, etc). L'opérateur '%' est l'opérateur modulo. Les opérateurs unaires '+' et '-' correspondent aux opérateur identité et opposé. Les opérateurs '++' et '--' sont des opérateurs permettant d'incrémenter (resp. décrémenter) une variable.

Les opérateurs de décalage permettent de décaler un entier d'un bit vers la gauche (<<) ou vers la droite (>>). L'operateur '~' est un opérateur de négation bit à bit. Le '&' est un opérateur réalisant un ET bit à bit, le '^' réalise un OU EXCLUSIF bit à bit et le '|' un OU bit à bit.

Les opérateurs relationnels sont utilisés dans des tests pour vérifier si deux variables sont égales ('=='), différentes ('!=') ou si la première est inférieure strictement à la deuxième ('<'), etc.

A noter que (type) est un opérateur de conversion. Exemple :

```
1 int n=3;
2 float x=1/(float)n;
```

Dans cet exemple, `n` est transformé en un nombre réel avant de faire le calcul. Si le `(float)` est enlevé, c'est une division entière qui est effectuée et `x` vaut alors 0.

`sizeof` est un opérateur donnant la taille mémoire occupée par une variable ou un type de donnée (exprimé en octets).

L'opérateur conditionnel permet de condenser un test et une affectation en une seule instruction. Exemple :

```
1 z=x>y?x:y;
```

Cette instruction compare `x` et `y`. Si `x` est supérieur, `z` prend la valeur `x`, sinon, il prend la valeur `y`. Cette instruction met donc dans `z` le maximum entre `x` et `y`.

Les opérateurs d'affectation attribuent une valeur à une variable. Lorsqu'un opérateur précède le signe '=', la variable reçoit le résultat de l'application de cet opérateur avec son ancienne valeur comme premier opérande et avec le membre de gauche de l'affectation comme deuxième opérande. Exemples :

```
1 x+=3;
2 /* equivalent a */
3 x=x+3;
4
5 x%=3;
6 /* equivalent a */
7 x=x%3;
```

2.7 STRUCTURES DE CONTRÔLE

Les structures de contrôle permettent de contrôler les instructions ou blocs d'instructions à exécuter. Elles sont essentiellement de deux types :

— conditionnel (`if`, `switch`) : choisit les instructions à exécuter en fonction d'une condition donnée

— répétitive (`while`, `for`, `do ... while`) : permet de répéter un bloc d'instructions

Le `if` permet de choisir entre deux alternatives. Il suit la syntaxe suivante :

```
1  if (expression)
2   {
3      instructions1;
4   }
5  else
6   {
7      instructions2;
8   }
```

Si l'expression est vraie (en C, par convention, toute valeur différente de 0 est considérée comme vraie), le bloc d'instructions 1 est exécuté, dans le cas contraire le bloc d'instructions 2 est exécuté.

Exemple :

```
1    if (x==0)
2   {
3      printf("x␣vaut␣0\n");
4   }
5   else
6   {
7      printf("x␣ne␣vaut␣pas␣0\n");
8   }
```

Le `switch` permet de choisir entre un nombre d'alternatives plus élevé. Une expression à valeur entière (ATTENTION, le `switch` ne fonctionne qu'avec des entiers) permet ainsi de choisir quelles instructions exécuter. Un bloc d'instructions est donné entre accolades après le `switch` et le mot clé `case` suivi d'une valeur constante entière permet de choisir à quel point commencer l'exécution.

```
1  switch (expression)
2     {
3     case expression-constante1 : instructions1;
4     case expression-constante2 : instructions2;
5     default : instructions3;
6     }
```

Le cas `default` est celui qui est choisi si aucune autre alternative n'est valide.

Sans indication contraire, l'exécution commence au point donné et ira jusqu'à l'accolade fermante indiquant la fin du bloc. Le comportement généralement souhaité est ceci dit d'exécuter un certain bloc d'instructions et de s'arrêter avant le prochain `case`. Pour obtenir ce comportement, on ajoute un `break` à l'endroit où l'on souhaite quitter le bloc. La syntaxe généralement utilisée est donc la suivante :

```
switch (expression)
   {
   case expression-constante1 : instructions1;
                                break;
   case expression-constante2 : instructions2;
                                break;
   default : instructions3;
   }
```

Exemple :

```
int n;
/* initialisation de n */
/* ... */
switch (n)
   {
   case 1 :
      printf("n vaut 1\n");
      break;
   case 2 :
      printf("n vaut 2\n");
      break;
   default :
      printf("n ne vaut ni 1 ni 2\n");
   }
```

Le `while` permet de répéter une boucle tant qu'une expression est vraie. Sa syntaxe est la suivante :

```
while (expression)
   {
   instructions;
   }
```

Exemple :

```
int n=10;
while (n>0)
   {
```

```
4         printf("n=%d\n",n);
5         n--; /* equivalent a n=n-1 */
6     }
```

Lorsque l'on connait le nombre d'itérations à réaliser, il est plus pratique d'utiliser une boucle `for`. La syntaxe est la suivante :

```
1  for (expression1; expression2; expression3)
2      {
3      instructions;
4      }
```

Cette boucle peut s'écrire de façon équivalente avec un `while` :

```
1  expression1;
2  while (expression2)
3     {
4     instructions;
5     expression3;
6     }
```

`expression1` correspond généralement à l'initialisation d'un compteur, `expression2` au test permettant d'arrêter la boucle ou de la continuer et `expression3` sert à mettre à jour le compteur.

Exemple :

```
1  int i;
2  for (i=0; i<10;i++)
3      {
4      printf("i=%d\n",i);
5      }
```

Le dernier type de boucle est le `do ... while`. Il fonctionne comme un `while` au détail près que le premier test est réalisé après une première exécution des instructions données. La syntaxe est la suivante :

```
1  do
2     {
3     instructions;
4     }
5  while (expression);
```

Quelques instructions permettent d'influencer le déroulement de ces structures de contrôle :

— `break` : interrompt une boucle ou un `switch`, n'exécute pas le reste des instructions et quitte la boucle ou le `switch`

— `continue` : passe à l'itération suivante, n'exécute pas le reste des instructions du bloc et recommence au début du bloc

2.8 FONCTIONS

Les fonctions servent à regrouper des instructions dans un même bloc. L'objectif est de simplifier la lecture et la compréhension d'un programme et également de maximiser la réutilisation de code.

La déclaration d'une fonction commence par son prototype, qui spécifie la valeur de retour, le nom de la fonction et, entre parenthèses, les différents arguments. L'instruction `return` permet de quitter la fonction en renvoyant la valeur spécifiée juste après le `return`.

ATTENTION : l'argument de `return` doit être de même type que la valeur de retour spécifiée dans le prototype.

Exemple :

```
1  int factorielle(int m) {
2    int i,n=1;
3    for (i=2;i<=m;i++) {
4      n*=i;
5    }
6    return n;
7  }
8  int main() {
9    int n=factorielle(10);
10 }
```

Une fonction ne peut pas renvoyer de tableau. Si une fonction prend un argument de type tableau, il est inutile de spécifier sa taille, elle sera ignorée par le compilateur. Il n'y a aucun moyen de connaître la taille d'un tableau passé en argument. Soit elle doit être fixée par convention (en espérant que les utilisateurs de la fonction la respecte...), soit il faut ajouter un argument supplémentaire la précisant. Les explications de tout cela seront données plus tard, dans le chapitre 5.

Une fonction ne peut être utilisée qu'après déclaration de son prototype, déclaration qui peut être faite à n'importe quel moment, même si le contenu de la fonction n'est pas déclaré. Ce point sera vu plus en détail dans le chapitre 4.

Si une fonction n'a pas besoin de renvoyer une valeur, il faut indiquer `void` comme type de retour.

Que deviennent les modifications faites à un paramètre à l'intérieur de la fonction ?

Exemple :

```
1  void f(int n) {
2     n=n+1;
3  }
```

```
4 int main(void) {
5    int i=2;
6    printf("avant:␣i=%d",i);
7    f(i);
8    printf("apres:␣i=%d",i);
9 }
```

Le résultat de ces instructions est le suivant :

```
1        avant: i=2
2        apres: i=2
```

Les modifications de `n` restent locales à la fonction `f` car ce qui est transmis est une copie de l'argument, on parle de *passage de paramètre par valeur*.

Les tableaux sont, encore, un cas particulier. Si des cases d'un tableau passé en paramètre sont modifiées les modifications ne sont pas perdues. Cette caractéristique sera utilisée à plusieurs reprises dans le chapitre suivant et la raison de cela, ainsi qu'une autre façon de transmettre des paramètres autre que des tableaux tout en gardant les modifications seront présentées au chapitre 5.

3 CHAÎNES DE CARACTÈRES

3.1 CARACTÈRES

Il n'y a pas de types spécifiques aux caractères en C. Les caractères sont manipulés au travers du code qui leur est associé dans la table des caractères ASCII. Un caractère est donc représenté par un entier. La table des caractères ASCII contient 127 caractères. Un octet suffit donc pour représenter un caractère, il est ainsi d'usage d'utiliser un `char` pour les représenter (néanmoins, n'importe quel entier peut être utilisé pour représenter un caractère). Un octet permet de représenter 256 valeurs, or seuls 127 valeurs sont nécessaires. De nombreuses extensions ont été proposées pour utiliser les valeurs restantes afin de représenter des caractères qui n'étaient pas dans la table des caractères ASCII, par exemple des caractères accentués [1].

Du point de vue du C, un caractère est équivalent à un entier. Seul l'usage qui en est fait permet de différencier des entiers de caractères.

Un caractère seul peut s'écrire en C sous la forme dudit caractère entouré de guillemets simples, par exemple 'a' ou 'b' désignent les caractères *a* et *b*.

Par exemple :

```
1  char c='a';
2  printf("c= %c %d\n",c,c);
```

Donne :

```
1  c= a 97
```

Ces instructions ont stocké dans `char c` le code associé au caractère *a*, puis elles ont affiché la valeur de `c` d'abord comme un caractère grâce au code %c (ce qui a donné le 'a'), puis comme un entier grâce au code %d.

3.2 CHAÎNES DE CARACTÈRES

Une chaîne de caractères est représentée sous la forme d'un tableau de `char`. Par convention la fin de la chaîne est indiquée par le caractère portant le code '\0'. Une chaîne de caractère constante est encadrée par des guillemets doubles, par exemple "UE 2I001". Le premier argument d'un `printf` est toujours une chaîne de caractères.

1. Il s'agit de l'extension normalisée ISO-8859-1.

Exemple :

```
1 char message1[8]="bonjour";
```

ATTENTION : bien prendre en compte le caractère '\0' final dans la taille du tableau !

Cette initialisation est équivalente à

```
1 char message1[8]={'b','o','n','j','o','u','r','\0'};
```

ou encore à :

```
1 char message1[8]={98, 111, 110, 106, 111, 117, 114, 0};
```

Ces différents entiers sont les codes des caractères de "bonjour". Le dernier entier est 0, qui est, par définition, le code du caractère '\0'.

Comme pour un tableau classique, il est possible d'omettre la taille lors de la déclaration si le tableau est initialisé. Dans ce cas, la taille du tableau est égale au nombre de caractères spécifiés plus un pour le caractère '\0'.

Exemple :

```
1         char message2[]="bonjour";
```

Remarque : un tableau de `char` peut bien sûr contenir une chaîne de caractères inférieure à sa taille, exemple `char msg[20]="hello";`. Les caractères qui suivent le caractère '\0' sont ignorés.

Exemple :

```
1 char nom[]="david\0goliath";
2 printf("nom=%s\n",nom);
```

affiche :

```
1 nom=david
```

3.3 FONCTIONS MANIPULANT DES CHAÎNES DE CARACTÈRES

L'utilisation des fonctions décrites ci-dessous nécessite d'ajouter la ligne suivante au début du fichier source :

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

3.3.1 strlen

```
1 size_t strlen (const char *s);
```

Renvoie la longueur d'une chaîne de caractères (`size_t` est un entier non signé).

3.3.2 strcmp et strncmp

```
1 int strcmp (const char *s1, const char *s2);
2 int strncmp(const char *s1, const char *s2,
3             size_t n);
```

Ces fonctions servent à comparer des chaînes de caractères. En effet, `s1==s2` ne compare pas les chaînes de caractères, mais leur emplacement en mémoire. Nous reviendrons plus tard sur le pourquoi. `strcmp` compare les deux chaînes de caractères passées en argument. Elle parcourt ces chaînes jusqu'à rencontrer une différence, s'il y en a, et s'arrête lorsqu'elle rencontre le caractère '\0'. Elle renvoie 0 si les chaînes sont identiques (jusqu'au caractère '\0' les terminant), sinon, elle renvoie une valeur positive si `s1` est après `s2` dans l'ordre alphabétique et une valeur négative sinon.

`strncmp` compare au plus les `n` premiers caractères.

3.3.3 strcpy et strncpy

```
1 char *strcpy (char *dest, const char *src);
2 char *strncpy(char *dest, const char *src,
3               size_t n);
```

Ces fonctions permettent de copier le contenu de la chaîne `src` dans la chaîne `dest`.

ATTENTION : il est bien nécessaire de passer par ces fonctions pour recopier une chaîne de caractères. `dest=src` ne permet pas de recopier les chaînes de caractères, mais tente de manipuler les emplacements mémoire de ces chaînes. Ces fonctions renvoient `dest`.

`strcpy` recopie `src` jusqu'à rencontrer le caractère '\0'. `strncpy` recopie `src` jusqu'à rencontrer le caractère '\0' ou jusqu'à ce que `n` caractères aient été recopiés (attention, dans ce cas il n'est pas ajouté de '\0' final).

ATTENTION : il faut que la destination soit assez grande pour recevoir la copie, sans quoi l'exécution de cette instruction provoquera des problèmes, typiquement une erreur de segmentation (segfault) ou un comportement inattendu (par exemple des variables sont modifiées alors qu'elles n'auraient pas dû l'être).

3.3.4 strdup

```
1 char *strdup(const char *s1);
```

`strdup` est une fonction qui crée un tableau de caractères identique à l'argument `s1`. Nous reviendrons sur cette fonction dans la section 5.7.1.

3.3.5 Exemples

```
1  char msg1[100]="bonjour";
2  char msg2[100]="aurevoir";
3
4  printf("Longueur de msg1=%d\n",strlen(msg1));
5
6  if (strcmp(msg1,msg2)) {
7          printf("msg1 different de msg2\n");
8  }
9
10 strcpy(msg2,msg1);
11 printf("msg2=%s\n",msg2);
12
13 if (!strcmp(msg1,msg2)) {
14         printf("msg1 egal a msg2\n");
15 }
```

Ces instructions affichent :

```
Longueur de msg1=7
msg1 different de msg2
msg2=bonjour
msg1 egal a msg2
```

Remarque : le compilateur peut afficher un message de type "warning" (avertissement) pour la ligne contenant le `strlen`. Pour le supprimer, il suffit de transformer le `size_t` en un `int`, ce qui se fait de la façon suivante :

```
1  printf("Longueur de msg1=%d\n",(int)strlen(msg1));
```

3.4 ENTRÉES/SORTIES "STANDARD"

Cette section contient une description des principales fonctions permettant d'écrire ou de lire sur la sortie standard (l'écran) ou depuis l'entrée standard (le clavier).

3.4.1 printf

Nous avons déjà vu cette fonction à de multiples reprises. Elle permet d'afficher un message à l'écran et renvoie un entier correspondant au nombre de caractères écrits.

```
1 int printf(const char * format, ...);
```

Les principaux formats sont les suivants :
— `%d` : entier
— `%u` : entier non signé
— `%c` : caractère
— `%s` : chaîne de caractères
— `%f` : nombre réel

Il est possible de choisir plus précisément la façon dont une valeur va être affichée en donnant une précision. La précision prend la forme d'un caractère '.' suivi d'un chiffre juste après le caractère '%' et avant le code de format. Pour un entier, cela correspond au nombre minimum de chiffres affichés, pour une chaîne de caractères le nombre maximum de caractères affichés et pour un nombre réel, cela correspond au nombre de chiffres affichés après la virgule (arrondi). Exemples :

```
1 printf("%.4d\n",42);
2 printf("%.4f\n",1.234567);
3 printf("%.4s\n","message");
```

Cela donne l'affichage suivant :

```
0042
1.2346
mess
```

3.4.2 scanf

`scanf` permet de lire des données saisies au clavier. Le prototype de la fonction est :

```
1 int scanf(const char * format, ...);
```

le premier argument est un format. Les arguments suivants indiquent où stocker les données qui ont été lues.

ATTENTION : les variables où stocker les données lues doivent être précédées du caractère &. Nous verrons au chapitre 5 la raison de cela.

Exemple :

```
1   float x;
2   int n;
3   scanf("%f␣%d",&x,&n);
```

Arrivé sur cette instruction, le programme va se bloquer en attendant que des données soient saisies au clavier. Les variables `x` et `n` contiennent les valeurs tapées au clavier.

ATTENTION : lors de la lecture, seul ce qui est demandé est lu !

Exemple pathologique :

```
1 char c;
2 scanf("%c",&c);
3 printf("Char␣lu:␣%c\n", c);
4 scanf("%c",&c);
5 printf("Char␣lu:␣%c\n", c);
```

Que fait le programme ?

Il ne se bloque pas dans l'attente d'un caractère pour le second `scanf`. En effet, lors du premier `scanf`, deux caractères ont été saisis : le caractère entré au clavier, puis le caractère '\n' (introduit par le retour chariot).

La solution consiste à ajouter un espace avant le premier code de format :

```
1 scanf("␣%c",&c);
```

L'espace indique par convention que le `scanf` doit "consommer" tous les espaces, retour à la ligne et autre tabulation avant de lire quoi que ce soit...

Les codes de format sont les mêmes que pour `printf` :

— %d : entier
— %u : entier non signé
— %f : flottant
— %c : caractère
— %s : chaîne de caractères

ATTENTION : là encore, les chaînes de caractères représentent une exception. Parmi les différents formats mentionnés ici, seules les chaînes de caractères ne nécessitent pas de & devant le nom de la variable. Ainsi pour lire une chaîne de caractères, il faut écrire :

```
1 char s[100];
2 scanf("%s",s);
```

ATTENTION : si plus de 100 caractères sont tapés au clavier, il y aura un débordement mémoire et, probablement, une erreur dite de segmentation.

3.4.3 getchar et putchar

Il existe des fonctions plus simples qui permettent de lire et d'écrire un seul caractère à la fois. Il s'agit des fonctions `getchar` et `putchar` :

— `int getchar()` : lecture d'un caractère. Renvoie le code du caractère lu, `EOF` si problème. Équivalent à
`scanf("%c",&c_lu)`

— `int putchar(int c)` : écriture d'un caractère. Renvoie le code du caractère écrit, `EOF` si problème. Équivalent à `printf("%c",c_a_ecrire)`

ATTENTION : comme pour le `scanf`, la validation de la saisie en appuyant sur la touche Entrée entraine l'ajout d'un caractère dans le buffer des caractères en attente d'être lus par le programme. Il faut donc absolument vérifier le caractère lu et recommencer tant que ce caractère est un retour à la ligne (autrement dit le caractère '\n'). Cela donne les instructions suivantes :

```
1 int c;
2 do
3   c=getchar();
4 while (c=='\n');
```

Remarque : si on souhaite ignorer, en plus des retours à la ligne, les espaces et tabulations, il est possible d'utiliser la fonction `isspace` qui renvoie une valeur non nulle (donc vraie) pour ces caractères. La boucle de lecture devient alors :

```
1 #include <ctype.h>
2
3 int c;
4 do
5   c=getchar();
6 while (isspace(c));
```

À la sortie de la boucle, `c` contient le code d'un caractère lu depuis le clavier et qui n'est ni un espace, ni un retour à la ligne, ni une tabulation.

3.5 ENTRÉES/SORTIES DANS DES FICHIERS

Les fichiers sont manipulés au travers d'une structure de données nommée `FILE`[2]. Cette structure est initialisée par une fonction d'ouverture, puis transmise en argument à toutes les fonctions qui vont lire et écrire dans le fichier. Cela permet à ces fonctions de savoir de quel fichier il s'agit ainsi que là où en est la lecture ou l'écriture. Une autre fonction permet de fermer le fichier.

3.5.1 Ouverture/fermeture d'un fichier : `fopen/fclose`

`FILE *fopen(const char *nom_fichier, const char *mode)` : ouvre le fichier selon le mode indiqué. Le mode peut être "r" (pour "read",

2. Nous n'avons pas encore vu les structures, mais ce n'est pas nécessaire pour comprendre la manipulation de fichiers.

lire), si le fichier est ouvert en lecture, "w" (pour "write", écrire), s'il est ouvert en écriture. Un fichier qui n'existe pas déclenche une erreur (la valeur de retour est NULL) en cas d'ouverture en lecture. Dans le cas d'une ouverture en mode d'écriture, le fichier est créé s'il n'existe pas. Dans ce cas, la valeur NULL indique que le fichier n'a pas pu être créé (dossier n'existant pas, droits insuffisants...). Si le fichier existe déjà, il est effacé. Le mode d'ouverture "a" (pour "append", ajouter) est similaire à "w", sauf que si le fichier existe déjà, il n'est pas écrasé, tout ce qui est écrit est ajouté à la fin du fichier.

`int fclose(FILE *flux)` : vide éventuellement le tampon de lecture/écriture, libère la mémoire correspondante et ferme le fichier. Il est important de faire appel à cette fonction. Dans certains cas, il est possible que des données soient perdues si cette fonction n'est pas appelée.

3.5.2 Lecture et écriture "formatée"

La lecture et l'écriture dite formatée fonctionne comme les fonctions d'entrées/sorties vues précédemment. Elles ont des noms et des signatures similaires préfixés d'un 'f' et un argument de type `FILE *`. Cela donne les fonctions suivantes :

- `int fscanf(FILE *flux,const char *format, ...)` : lecture de données depuis un fichier à partir d'un format (même principe que `scanf`)
- `int fprintf(FILE *flux, const char *format,...)` : écriture de données dans un fichier selon un format spécifié (même principe que `printf`)
- `int fputc(int c, FILE *flux)` : écrit un caractère dans un fichier (même principe que `putchar`)
- `int fgetc(FILE *flux)` : lit un caractère depuis un fichier (même principe que `getchar`)
- `char * fgets(char *buffer, int n, FILE *flux)` : lit une ligne dans un fichier (longueur maximale=n) et met le résultat dans la chaîne de caractère `buffer`. Renvoie `NULL` si la fin du fichier est atteinte.

Exemple :

```
1  #include <stdio.h>
2
3  int main() {
4
5    FILE *src, *dest;
6    int car;
7
8    if ((src= fopen("source.txt","r"))==NULL) {
9      printf("Erreur␣a␣l'ouverture␣de␣source.txt\n");
10     return 0;
```

```
11    }
12    if ((dest= fopen("destination.txt","w"))==NULL) {
13      printf("Erreur␣a␣l'ouverture␣de␣dest.txt\n");
14      return 0;
15    }
16    car = fgetc(src);
17    while (car!=EOF) {
18      fputc(car,dest);
19      car = fgetc(src);
20    }
21    fclose(src);
22    fclose(dest);
23    return 0;
24  }
```

Cette fonction lit le contenu d'un fichier nommé "source.txt" caractère par caractère et l'écrit dans un fichier nommé "destination.txt". Notez la valeur spéciale `EOF`, qui est une constante renvoyée par les fonctions de lecture lorsque la fin du fichier est atteinte (EOF signifie End Of File). Il est également possible d'utiliser la fonction `int feof(FILE *stream)` qui renvoie 0 (donc faux) si la fin du fichier n'est pas atteinte et une valeur non nulle (donc vraie) sinon.

L'exemple précédent correspond à une lecture caractère par caractère. Cette façon de lire un fichier n'est pas toujours des plus pratiques. Un fichier formaté étant écrit ligne par ligne, il est souvent plus pratique de le lire de cette façon.

L'exemple suivant détaille la lecture d'un fichier ligne par ligne. Chaque ligne est affichée à l'écran avec un `printf`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main() {
6
7    char buffer[256];
8    char *res;
9
10   /* a remplacer par n'importe quel nom de fichier
11   dans le repertoire courant */
12   FILE *f=fopen("fichier_a_lire.txt","r");
13
14   if (f == NULL) {
15     printf ("Probleme\n");
16     return 1;
```

```
17    }
18
19    /* on lit la premiere ligne */
20    res = fgets(buffer, 256, f);
21
22    /* tant que la fin du fichier n'est pas atteinte */
23    while(res!=NULL) {
24
25       /* on affiche la ligne lue */
26       printf("Ligne␣lue:␣%s",res);
27
28      /* on lit la ligne suivante */
29      res = fgets(buffer, 256, f);
30    }
31
32    /* on ferme le fichier */
33    fclose(f);
34
35    return 0;
36  }
```

Dans un fichier formaté, une ligne pourra contenir des informations que l'on souhaite récupérer (par exemple des valeurs entières ou des nombres réels). L'interprétation d'une ligne sans connaître son contenu est un problème très important en informatique, mais qui dépasse largement les limites de ce cours (c'est une des questions à résoudre pour faire un compilateur...). Seul le cas où le contenu d'une ligne est connu sera abordé ici. Dans ce cas, le contenu d'une ligne peut être récupéré à l'aide de la fonction `sscanf`, dont le fonctionnement est similaire à `scanf`, sauf que `sscanf` lit les données dans la chaîne de caractères transmise dans le premier argument :

```
1  int sscanf(const char * s, const char * format, ...);
```

Dans l'exemple suivant, chaque ligne contient trois entiers qui sont placés dans les variables `a`, `b` et `c` avant d'être affichés à l'écran.

```
1  #include <stdio.h>
2  #define LONGUEURLIGNE 128
3   int main() {
4     FILE *src;
5     if ((src= fopen("source.txt","r"))==NULL) {
6       printf("Erreur␣a␣l'ouverture␣de␣source.txt\n");
7       return 1;
8     }
9     char buffer[LONGUEURLIGNE];
```

```
10    char *res=fgets(buffer, LONGUEURLIGNE, src);
11    int a,b,c;
12    while (res != NULL) {
13       sscanf(buffer,"␣%d␣%d␣%d",&a, &b, &c);
14       printf("Lu␣les␣entiers␣a=%d,␣b=%d,␣c=%d\n",a,b,c);
15
16       res=fgets(buffer, LONGUEURLIGNE, src);
17    }
18
19    fclose(src);
20    return 0;
21 }
```

Remarque : nous verrons un autre mode de lecture/écriture, la lecture/écriture binaire, dans le chapitre 6.

Les fonctions de lecture/écriture dans des fichiers peuvent être utilisées sur des descripteurs correspondant à l'entrée standard (nommée `stdin`, il s'agit en général du claiver), à la sortie standard (nommée `stdout`, il s'agit en général du terminal) et à la sortie d'erreur (nommée `stderr`, il s'agit en général également du terminal). Exemple :

```
1 char buffer[128];
2 fgets(buffer,128,stdin);
3 fprintf(stdout,"Affichage␣sur␣la␣sortie␣standard:%s\n",
4         buffer);
5 fprintf(stderr,"Affiche␣sur␣la␣sortie␣d'erreur:␣%s\n",
6         buffer);
```

Remarque : par défaut, la sortie standard et la sortie d'erreur sont affichées toutes deux dans le terminal. Il est possible de les séparer, par exemple de rediriger la sortie d'erreur vers un fichier spécifique[3].

3.6 ARGUMENTS DE LA FONCTION MAIN

Jusqu'à présent, nous avons défini des fonctions `main` sans argument (`void`). Il existe un autre prototype de la fonction main avec deux paramètres. Ces paramètres permettent de récupérer les arguments transmis au programme lors de son exécution. Ces arguments sont des chaînes de caractères. Exemple :

```
./mon_prog arg1 arg2 arg3
```

3. Si le programme s'appelle prog, pour rediriger la sortie d'erreur vers le fichier erreurs.log en laissant l'affichage de la sortie standard à l'écran : `./prog 2>erreurs.log`.

Dans cet exemple, le programme `mon_prog` reçoit les arguments `arg1`, `arg2` et `arg3`.

ATTENTION : par convention, la chaîne de caractères utilisée pour appeler le programme (ici `./mon_prog`) est considérée comme le premier argument transmis au programme. Dans l'exemple, le programme reçoit donc quatre arguments.

Le prototype de `main` permettant de récupérer les arguments transmis dans le terminal est le suivant :

```
1 int main (int argc, char *argv[]);
```

`argc` vaut le nombre d'arguments (en comptant la chaîne utilisée pour appeler le programme) et `argv` est un tableau de chaînes de caractères (voir le chapitre sur les pointeurs pour mieux comprendre son type). Exemple d'utilisation :

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[]) {
4   unsigned int i;
5   for (i=0;i<argc;i++) {
6     printf("Argument␣%d:␣%s\n",i,argv[i]);
7   }
8   return 1;
9 }
```

Exemple d'appel :

```
$ ./mon_prog arg1 arg2 arg3
Argument 0: ./mon_prog
Argument 1: arg1
Argument 2: arg2
Argument 3: arg3
```

4 DÉCOUPAGE, COMPILATION ET DÉBOGAGE

Ce chapitre présente les principes de la compilation ainsi que le découpage d'un programme en plusieurs fichiers. Il présente également les macros ainsi que les qualifieurs de variables. Les outils issus du monde du logiciel libre pour la compilation et du débogage sont ensuite évoqués. Ils ont l'avantage d'être gratuits et disponibles sous toutes les distributions Linux, sous MacOS X ainsi que sous Windows au travers de CygWin ou MinGW et ils seront utilisés pendant les séances de TME.

4.1 DÉCOUPAGE D'UN PROGRAMME

Lorsqu'un programme nécessite de nombreuses lignes de codes, il devient plus pratique de le découper en plusieurs fichiers séparés afin de simplifier son édition. Un tel découpage permet de créer une certaine modularité du code en regroupant dans un fichier toutes les fonctions en charge d'une fonctionnalité particulière. Si cet ensemble de fonctions peut servir à un autre programme, il suffit alors de réutiliser le fichier source correspondant. Nous allons également voir que, si ces fonctions ne sont pas modifiées, le compilateur n'aura pas besoin de les recompiler pour constituer un nouvel exécutable, réduisant ainsi le temps de compilation. Si la durée de compilation est négligeable pour un programme court, elle peut prendre plusieurs minutes, voire plus sur des programmes conséquents.

En C, il existe par convention deux types de fichiers source : les fichiers avec l'extension ".c" et les fichiers avec l'extension ".h" (appelé fichiers entête ou *header* en anglais). Les fichiers ".c" sont ceux qui contiennent la déclaration des fonctions, donc le principal contenu d'un programme (dont la fonction main). Les fichiers headers ne contiennent pas de déclaration de fonctions, mais des prototypes de fonction, des déclarations de type ou de variables globales [1].

Pour avoir le droit d'utiliser une fonction dans un fichier source, il faut que le compilateur, au moment ou il va lire et analyser le programme, connaisse son prototype. Si cette fonction a été déclarée avant dans le même fichier source, son prototype sera connu. Si la fonction est déclarée dans un

1. Ces déclarations sont précédées du qualifieur `extern`, ce point sera évoqué plus loin dans ce chapitre.

autre fichier source, il faut déclarer son prototype (à savoir son type de retour, son nom et, entre parenthèses, la liste de ses arguments) suivi d'un ';' dans le fichier source avant de l'utiliser (typiquement au début du fichier). Cependant, cela pourrait conduire à recopier à de multiples reprises ce prototype avec toutes les difficultés que cela peut poser si jamais on souhaitait le modifier (il faudrait alors changer tous les endroits où le prototype a été déclaré). C'est pour éviter ces problèmes qu'ont été définis les fichiers entête : ils contiennent les prototypes de fonctions et peuvent être inclus dans n'importe quel fichier source à l'aide de la macro "#include" qui sera présentée plus tard. Le prototype est ainsi décrit ici, puis inclus partout où il doit être connu. Ce principe a déjà été utilisé précédemment pour pouvoir utiliser les fonctions les plus courantes du C :

— le prototype de `printf` est déclaré dans `stdio.h` ;
— les prototypes des fonctions de manipulation des chaînes de caractères (`strlen`, `strcpy`, `strcmp`, ...) sont déclarés dans `string.h`.

Si l'on souhaite de même utiliser une variable globale dans une fonction, elle doit être connue du compilateur au moment où il analysera la fonction en question. Soit elle a été déclarée avant dans le même fichier source et elle est alors connue, soit il faut la faire connaitre. Si on se contente de la redéclarer au début du fichier source, le compilateur créera une nouvelle variable et il y aura un conflit lors de la création de l'exécutable. Il faut donc indiquer que cette variable existe ailleurs, sans la créer. Cela se fait avec le qualifieur `extern`, qui sera présenté plus tard. Ainsi la ligne suivante :

```
1 extern int ma_variable_globale;
```

indique au compilateur qu'une variable appelée `ma_variable_globale` de type `int` existe "quelque part" et que l'on peut l'utiliser.

Exemple :
Fichier mes_fonctions.h :

```
1 extern float ma_variable;
2 int ma_fonction1(int, float);
3 void ma_fonction2(float, char[10]);
```

Fichier mes_fonctions.c :

```
1 float ma_variable=12.;
2 int ma_fonction1(int i, float x) {
3    /* contenu de la fonction ... */
4 }
5 void ma_fonction2(float y, char t[10]) {
6    /* contenu de la fonction ... */
7 }
```

Fichier mon_programme.c, utilisant les fonctions définies dans mes_fonctions.c :

```
1  #include "mes_fonctions.h"
2
3  int main() {
4    int i=0,j;
5    float f=ma_variable;
6    j=ma_fonction1(i,f);
7
8    /* suite du programme ...*/
9
10   return 0;
11 }
```

ATTENTION : l'exemple ci-dessus est fonctionnel, cependant, les fichiers en-tête que vous rencontrerez seront toujours de la forme :

```
1 #ifndef _NOM_DU_FICHIER_H_
2 #define _NOM_DU_FICHIER_H_
3
4 /* contenu du fichier en-tete */
5
6 #endif
```

nous verrons pourquoi dans la section 4.3.3.

4.2 COMPILATION

La compilation permet de passer d'un (ou de plusieurs) fichiers texte contenant le code C écrit par le programmeur à un fichier exécutable que le processeur de l'ordinateur pourra exécuter pour obtenir le résultat attendu. Il s'agit en fait d'un processus de "traduction" : le langage C, compréhensible pour les humains (du moins en théorie) est traduit en un langage machine, propre au processeur de l'ordinateur sur lequel on souhaite exécuter le programme. Ce langage machine contient une succession d'instructions élémentaires pour manipuler la mémoire et les registres de l'ordinateur. La moindre boucle ou fonction nécessite l'écriture de nombreuses instructions alors que leur écriture est très simple en C. C'est ce qui motive le développement de langages autres que ce langage machine [2]. Le rôle des compilateurs est de transformer le ou les fichiers sources en un programme en langage machine aussi rapide que possible.

La compilation en C se déroule en plusieurs étapes :

2. D'autres langages que le C sont d'encore plus haut niveau et permettent d'écrire de façon relativement simple et compacte des programmes qui peuvent être très complexes.

1. Traitement de chaque fichier source indépendamment :
   — prétraitement : gestion des macros et autres directives au préprocesseur
   — compilation : transformation du code obtenu en un fichier objet
2. Édition des liens entre les fichiers objets pour générer l'exécutable.

La première étape se fait indépendamment sur chaque fichier source (.c). Elle consiste à le traduire en langage machine. Le fichier résultant, appelé fichier objet (avec l'extension ".o") contient toutes les fonctions contenues dans le fichier ".c" traduites en langage machine. Ce fichier contient également des symboles non définis, il s'agit des fonctions qui ne sont pas déclarées dans ce fichier source (qui n'étaient connues qu'au travers de leur prototype) ou des variables globale qui n'étaient pas non plus déclarées dans ce fichier source. Il est à noter que cette étape est décomposées en deux étapes distinctes, une étape de prétraitement pendant laquelle sont interprétées toutes les macros et ensuite l'étape de compilation à proprement parler. D'un point de vue schématique, après cette étape on se retrouve avec des fichiers "à trous", ne contenant qu'une partie du programme.

La seconde étape consiste à mettre ensemble tous ces fichiers objets et à boucher ces trous. Une fonction utilisée dans un fichier source doit toujours être déclarée quelque part[3]. S'il ne manque aucune pièce à ce puzzle à l'issue de cette étape d'édition de lien, le fichier exécutable est généré. Si une fonction ou une variable reste inconnue ou bien si elle est décrite à plusieurs reprises dans des fichiers source distincts, alors l'éditeur de lien renvoie une erreur et ne peut créer l'exécutable. Dans le cas de fonctions ou variables manquantes, l'erreur fera référence à des symboles non définis ("Undefined symbols"), dans le cas de symboles définis plusieurs fois, l'erreur fera référence à des symboles dupliqués ("Duplicate symbols").

`gcc` réalise ces deux opérations. Par défaut, il les fait directement toutes les deux, c'est pourquoi l'exemple précédent :

```
gcc -o welcome welcome.c
```

permettait de créer l'exécutable à partir du fichier source "welcome.c". Pour lui indiquer de s'arrêter après la première étape, il faut ajouter l'option "-c". Ainsi

```
gcc -c -o welcome.o welcome.c
```

crée le fichier objet "welcome.o", puis :

```
gcc -o welcome welcome.o
```

crée l'exécutable à partir du fichier objet.

L'exemple présenté plus haut avec deux fichiers source peut se compiler de la façon suivante :

3. Les fonctions standards du C sont disponibles dans la bibliothèque standard du C qui est incluse par défaut.

1. prétraitement et compilation des sources :

```
gcc -c -o mes_fonctions.o mes_fonctions.c
gcc -c -o mon_programme.o mon_programme.c
```

2. édition des liens :

```
gcc -o mon_programme mon_programme.o mes_fonctions.o
```

Remarque 1 : le fichier en-tête "mes_fonctions.h" n'apparaît nulle part ici. C'est tout à fait normal, il sera pris en compte via les macros `#include` et n'apparaitra jamais dans une commande de compilation.

Remarque 2 : sous linux, il est possible de consulter la table des symboles en utilisant la commande "nm" dans un terminal. Sur les fichiers objets crées précédemment, elle donne le résultat suivant :

```
nm mes_fonctions.o
0000000000000028 s EH_frame1
0000000000000000 T _ma_fonction1
0000000000000040 S _ma_fonction1.eh
000000000000000e T _ma_fonction2
0000000000000070 S _ma_fonction2.eh
0000000000000020 D _ma_variable


nm mon_programme.o
0000000000000038 s EH_frame1
                 U _ma_fonction1
                 U _ma_variable
0000000000000000 T _main
0000000000000050 S _main.eh
```

cette commande indique les symboles présents dans les fichiers objets avec, éventuellement leur valeur avec une lettre indiquant leur type. Tout ce qui nous intéresse ici, c'est de voir quels symboles sont définis et quels symboles ne sont pas définis. Les symboles précédés de 'U' et pour lesquels aucune valeur n'est donnée ne sont pas définis, tous les autres sont définis [4]. On voit apparaître les fonctions et variables utilisées dans mon_programme.c et non définies dans ce fichier source, à savoir `ma_fonction1` et `ma_variable`. On voit également que mes_fonctions.o ne contient aucun symbole indéfini et qu'il contient bien une définition des symboles `ma_fonction1` et `ma_variable`. On peut donc vérifier ici qu'il n'y aura pas de "trous" lorsque les deux fichiers objets seront assemblés.

4. faire `man nm` dans un terminal pour plus de détails sur ce qu'affiche cette commande.

4.3 MACROS

Les macros sont des instructions spécifiques qui sont prises en compte par le préprocesseur avant la compilation. Il s'agit d'instructions commençant par #.

4.3.1 #include

La macro `#include` qui a déjà été évoquée à plusieurs reprises permet d'inclure un fichier en-tête. Ce type de fichier contient, par exemple, des déclarations de prototypes de fonctions. Cela permet ensuite d'utiliser toutes les fonctions, variables globales ou types qui peuvent être déclarés à l'intérieur de ce fichier. En pratique, cette macro correspond à un copié-collé du fichier spécifié en argument à l'emplacement de ladite macro. L'argument peut être donné entre chevrons (<>) ou entre guillemets doubles (""). Dans le premier cas, le compilateur ira chercher le fichier dans les répertoire "système" (par exemple /usr/include). Dans le cas des guillemets doubles, le fichier sera cherché dans le répertoire courant. Exemples :

```
1 #include "mes_fonctions.h"
2 #include <stdio.h>
```

4.3.2 #define

La macro `#define` permet de remplacer une chaîne par une autre. En pratique, cette macro est souvent utilisée pour définir des constantes. Exemple :

```
1 #define TAILLE 10
```

Après cette macro, chaque fois que le compilateur rencontrera la chaîne de caractères TAILLE, il la remplacera par la chaîne de caractères 10.

ATTENTION : il s'agit bien là d'un remplacement tel que vous le feriez avec les fonctions de copié-collé de votre éditeur. Si vous vous trompez que vous ajoutez un ';' à la fin de la ligne, TAILLE sera alors remplacé par 10; partout dans le programme, ce qui provoquera des erreurs de compilation qui ne seront pas forcément aisées à comprendre.

Remarque : il est possible de définir des pseudo-fonctions avec cette macro. Exemple :

```
1 #define CARRE(X) (X)*(X)
```

Chaque fois que le compilateur rencontrera CARRE(quelque chose) il le remplacera par (quelque chose)*(quelque chose).

ATTENTION : il ne s'agit pas là d'une fonction, mais toujours d'un copié-collé. En particulier, les arguments ne sont pas calculés avant de d'exécuter

l'instruction de droite. C'est pourquoi il faut bien mettre des parenthèses dans la partie droite, sinon, le résultat risque de ne pas être celui qui est attendu. Exemple : CARRE(1+1) sera bien remplacé par (1+1)*(1+1). Si on avait omis les parenthèses, il serait remplacé par 1+1*1+1, ce qui ne donnerait pas le comportement attendu... Cette possibilité vous est donnée pour mémoire, ce type d'écriture étant parfois utilisé en TD. Elle doit cependant être utilisée avec beaucoup de précautions car elle peut provoquer des erreurs difficiles à trouver, c'est la raison pour laquelle elle n'est pas recommandée, à moins que vous ne sachiez avec certitude ce que vous faites.

4.3.3 `#ifdef #ifndef`

Les macros `#ifdef` et `#ifndef` permettent de prendre en compte ou d'ignorer une partie d'un fichier en fonction de l'existence d'une étiquette qui aurait été définie (ou pas) auparavant par un `#define`.

`#ifdef` ... `#endif` ou `#ifdef` ... `#else` ... `#endif` indiquent un ensemble de lignes à inclure si l'étiquette spécifiée après le `#ifdef` est définie. Si elle n'est pas définie, l'ensemble de lignes spécifiées entre le `#else` et le `#endif` sera inclus si il existe (sinon, rien ne sera inclus). Exemple :

```
1 #define ALTERNATIVE1
2
3 #ifdef ALTERNATIVE1
4 /* contenu de l'alternative 1 */
5 #else
6 /* contenu a inclure si l'etiquette ALTERNATIVE1
7     n'est pas definie */
8 #endif
```

`#ifndef` fonctionne de la même manière, mais en négatif, il indique qu'il faut inclure les lignes spécifiées si l'étiquette n'est pas définie.

Ces macros sont souvent utilisées pour décrire des parties de code qui pourraient être spécifique à une architecture de processeur particulière ou pour traiter à la compilation de cas particuliers (outre le `#define`, il est possible de définir des étiquette à la compilation). Dans la suite de ce cours, nous ne reviendrons pas sur cet usage. Par contre, ces macros apparaitront systématiquement dans les fichiers en-tête que nous allons définir dans la suite. Tous seront de la forme :

```
1 #ifndef _NOM_DU_FICHIER_H_
2 #define _NOM_DU_FICHIER_H_
3 /* contenu du fichier en-tete */
4 #endif
```

Ces macros testent l'existence d'une étiquette, puis, si elle n'existe pas, elle est définie. Si elle existe déjà, tout ce qui est entre le `#ifndef` et le `#endif` est tout simplement ignoré.

Pourquoi encadrer tout le contenu des fichiers en-tête par ces macros ? Pour des programmes un peu complexes, un fichier en-tête peut se retrouver inclus à plusieurs reprises dans un même fichier source (si le même fichier en-tête est lui-même inclus par plusieurs en-tête qui sont eux-même inclus dans le fichier source, par exemple). Dans ce cas de figure, si le fichier ne contient que des prototypes de fonctions ou des déclarations de constantes précédées du mot clé `extern`, ce n'est pas grave. Par contre, si le fichier contient une déclaration de type (nous verrons comment faire au chapitre 6), l'inclure plusieurs fois provoquera une erreur de compilation. Pour éviter ces problèmes, on prendra le réflexe de toujours ajouter ces macros, même si elles ne sont pas toujours indispensables. L'étiquette à utiliser peut être quelconque, ce qui compte c'est qu'elle soit unique, c'est pourquoi on la définit en général à partir du nom du fichier.

4.4 QUALIFIEURS

Les qualifieurs sont des mots-clés qui sont ajoutés avant le nom de type d'une variable pour ajouter une information à destination du compilateur (ou de la personne qui va lire le code). Tous les qualifieurs ne sont pas liés au découpage d'un programme, mais certains le sont. C'est pour cette raison que nous avons regroupé leur présentation dans ce chapitre.

4.4.1 `extern`

Le mot-clé `extern` indique qu'une variable globale existe et a été déclarée dans un autre fichier source. Le compilateur ne créera donc pas la variable en question, mais elle sera tout de même considérée comme connue dans la suite (c'est l'équivalent à la déclaration du prototype pour une fonction).

Exemples :

```
1  #include <stdio.h>
2
3  int ma_variable1;
4  extern int ma_variable2;
5
6  int main(foid) {
7    printf("ma_variable1=%d\n",ma_variable1);
8    printf("ma_variable2=%d\n",ma_variable2);
9
10   return 0;
```

```
11 }
```

`ma_variable1` est une variable globale déclarée dans le fichier source donné ci-dessus. `ma_variable2` est aussi une variable globale, mais qui a été déclarée dans un autre fichier source. Les deux peuvent être lues ou modifiées de la même manière.

ATTENTION : la variable `ma_variable2` doit être déclarée dans un autre fichier source (sans le mot-clé extern).

Remarque 1 : une déclaration de variable globale précédée du mot-clé `extern` peut apparaître autant de fois que nécessaire, par contre, elle ne peut (et doit) apparaître qu'une seule fois sans ce mot-clé.

Remarque 2 : il est interdit d'initialiser une variable déclarée comme `extern` :

```
1 int ma_variable1= 42; /* AUTORISE */
2 extern int ma_variable2 = 24; /* INTERDIT !*/
```

4.4.2 static

Le mot clé `static` est utilisé dans deux contextes différents :

1. devant une variable globale ou une fonction : elle ne sera "visible" que depuis le fichier source dans lequel elle est déclarée. Seule une fonction de ce fichier pourra y accéder.
2. devant une variable locale à une fonction : la variable ne sera initialisée qu'une fois et conservera sa valeur d'un appel à un autre. On parle de classe d'allocation statique.

Exemple d'utilisation pour une variable globale : Fichier mes_fonctions1.c :

```
1 int n=3;
2 static int m=4;
```

Fichier mes_fonctions2.c :

```
1 extern int n; /* permet d'acceder
2 au n defini dans mes_fonctions.c */
3 extern int m; /* INTERDIT ! */
4 /* m ne peut etre connu que dans mes_fonctions1.c  */
5
6 void ma_fonction(void) {
7   n=6;
8   /* ... */
9 }
```

Exemple d'utilisation pour une variable locale :

```
1 void ma_fonction(void) {
2   static int compteur=0;
3   compteur++;
4   printf("Ma␣fonction␣a␣ete␣appelee␣%d␣fois",compteur);
5 }
```

4.4.3 const

Le mot clé `const` permet de déclarer une variable en indiquant que son contenu ne changera pas. Exemple :

```
1     const int i=3;
2     const float x=2.*3.1415926;
```

Ce mot clé peut aussi être utilisé avec des arguments de fonction. Exemple :

```
1 void f(const int n, const char t[10]);
```

Le `const` indique que ni `n`, ni le contenu du tableau `t` ne sera changé dans la fonction (dans le cas contraire, le compilateur indiquera une erreur de compilation). Cela indique aux programmeurs qui utiliseront cette fonction que ces arguments ne seront pas modifiées. Pour `n`, les modifications étant locales, cela ne changera pas grand chose, par contre, pour `t`, comme il s'agit d'un tableau, toute modification réalisée dans `f` aurait changé le tableau et non une copie locale. Il s'agit donc là d'une information qui peut être importante pour tout utilisateur de cette fonction.

Ce mot clé est pris en compte pendant la compilation, si le compilateur détecte une modification d'une variable constante, il indiquera une erreur de compilation.

4.5 OUTILS

4.5.1 GCC

`gcc` est un compilateur C libre et gratuit (http://gcc.gnu.org/). Il est disponible pour des systèmes d'exploitation variés (linux, MacOS X ou Windows via CygWin ou MinGW). Il doit être utilisé depuis un terminal de la façon suivante :

```
gcc [options] source1.c source2.c...
```

Les options les plus couramment utilisées sont les suivantes :

- `-c` : ne pas faire l'édition de liens

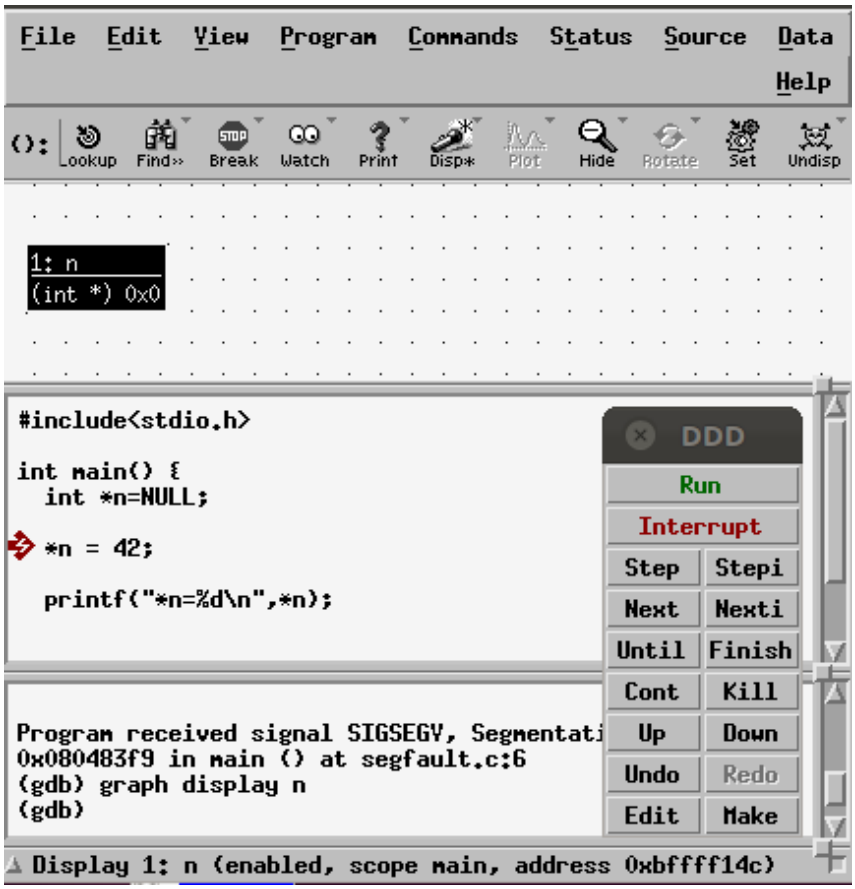


FIGURE 1 – Interface graphique de DDD.

- `-o fichier_sortie` : nom du fichier de destination (fichier .o ou exécutable selon les cas). Si non spécifié, `a.out` pour un exécutable, `source.o` pour le fichier objet associé à un fichier `source.c`.
- `-Wall` : affiche des "warnings", messages indiquant non pas des erreurs, mais des instructions suspectes. Il est recommandé d'utiliser systématiquement cette option et de prendre en compte tous les warnings, cela peut faire gagner du temps sur le débogage ;
- `-g` : inclure les informations de débogage (nécessaire pour utiliser un débogueur).

Pour information :

- `-E` : ne fait que le prétraitement et envoie le résultat sur la sortie standard.

4.5.2 GDB & DDD

Un débogueur est un logiciel permettant d'observer le comportement d'un programme pendant son exécution. On peut l'exécuter en pas-à-pas, c'est-à-

dire instruction par instruction, ou bien exécuter le programme jusqu'à une instruction particulière, etc. Il est possible d'afficher la valeur de variables. En cas d'erreur à l'exécution, la ligne incriminée est pointée. Un tel programme est très pratique pour comprendre comment marche un exécutable et pour trouver la source d'erreurs.

`gdb` est un débogueur libre et gratuit (http://www.gnu.org/software/gdb/). Il est très puissant, mais fonctionne en ligne de commande directement dans le terminal. Des interfaces graphiques à `gdb` ont été développées pour proposer une interface plus conviviale. `ddd` en est une (http://www.gnu.org/software/ddd/, figure 1).

Pour être débogué, un programme doit être compilé avec l'option `-g`. Le déboguage d'un programme nommé `mon_prog` est lancé dans le terminal avec la commande suivante :

```
ddd ./mon_prog
```

L'exécution de cette commande ouvre la fenêtre apparaissant sur la figure 1. Pour lancer l'exécution du programme, il suffit de cliquer sur "Run"[5]. En cas d'erreur de segmentation la ligne en cause apparaît avec une flèche (figure 1). Il est possible d'afficher la valeur d'une variable en cliquant sur son nom dans la fenêtre représentant le code source avec le bouton gauche. Beaucoup d'autres utilisations sont possibles, que le programme soit bogué ou pas. Pour mieux comprendre le fonctionnement d'un programme, il est possible de poser des points d'arrêt pour que l'exécution s'arrête sur cette ligne, il suffit pour cela de cliquer sur le début de la ligne où installer le point d'arrêt avec le bouton droit et de choisir "Set Breakpoint".

La documentation de DDD est disponible ici : http://www.gnu.org/software/ddd/manual/ (en anglais).

4.5.3 Valgrind

`valgrind` n'est pas un débogueur à proprement parler, mais il est très efficace pour détecter toute manipulation suspecte de mémoire (http://valgrind.org/). Il permet également de détecter les fuites mémoire, l'utilisation de variables non initialisées, etc. Il s'agit aussi d'un logiciel libre et gratuit, disponible sous linux et, plus récemment sous Mac OS X.

Son utilisation est très simple : il suffit d'appeler l'exécutable dans le terminal en ajoutant valgrind devant son nom. Valgrind ajoutera un certain nombre de messages dans le terminal pour indiquer ce qu'il a trouvé de suspect. Il est recommandé de tester ses exécutables avec valgrind. Même en l'absence de plantage du programme, il peut détecter des comportements suspects et donc grandement faciliter le débogage.

A titre d'exemple, le programme suivant :

5. Si la fenêtre contenant toutes les commandes n'apparaît pas, il suffit de cliquer sur View->Command Tool.

```
1 #include <stdio.h>
2
3 int main(void) {
4   int *p=NULL;
5
6   *p=42;
7   printf("*p=%d\n",*p);
8   return 0;
9 }
```

appelé avec Valgrind donne le resultat suivant (le programme s'appelle `mon_segfault` et le fichier source `mon_segfault.c`) :

```
$ valgrind ./mon_segfault
==2507== Memcheck, a memory error detector
==2507== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==2507== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==2507== Command: ./mon_segfault
==2507==
==2507== Invalid write of size 4
==2507==    at 0x4004C4: main (mon_segfault.c:6)
==2507==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==2507==
==2507==
==2507== Process terminating with default action of signal 11 (SIGSEGV)
==2507==  Access not within mapped region at address 0x0
==2507==    at 0x4004C4: main (mon_segfault.c:6)
==2507==  If you believe this happened as a result of a stack
==2507==  overflow in your program's main thread (unlikely but
==2507==  possible), you can try to increase the size of the
==2507==  main thread stack using the --main-stacksize= flag.
==2507==  The main thread stack size used in this run was 8388608.
==2507==
==2507== HEAP SUMMARY:
==2507==     in use at exit: 0 bytes in 0 blocks
==2507==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==2507==
==2507== All heap blocks were freed -- no leaks are possible
==2507==
==2507== For counts of detected and suppressed errors, rerun with: -v
==2507== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
```

La première ligne indique la façon d'appeler le programme (attention à ne pas oublier le "./" ou tout du moins le chemin pour trouver l'exécutable). Les quatre premières lignes donnent le numéro de version et diverses informations générales.

Les lignes "Invalid write of size 4" indiquent l'endroit où l'erreur a eu lieu et la raison : le programme a tenté d'écrire 4 octets (un `int`) à l'adresse `0x0`, ce qui n'est pas autorisé. Cette erreur a été déclenchée par la ligne 6 du fichier source (mon_segfault.c :6). La suite donne des recommandations sur une cause possible de ce problème (débordement mémoire, ce n'est pas le problème ici). Il fait ensuite un bilan de l'utilisation mémoire.

4.5.4 Makefile

La compilation est composée de deux étapes : une qui doit être réalisée séparément sur chaque fichier source et une qui va les "assembler" pour créer l'exécutable. Pour éviter d'avoir à recopier les commandes de compilation, il existe des outils permettant l'automatisation de ce processus.

`make` est un de ces outils, qui permet aussi de ne recompiler un fichier source que si c'est nécessaire, ce qui peut faire gagner beaucoup de temps pour des programmes contenant de nombreuses lignes de code. C'est un outil utilisé très fréquemment dans le monde du logiciel libre. `make` est un outil très puissant dont la présentation dépasse largement le cadre de ce cours. Nous allons n'en voir qu'une infime partie, mais qui est suffisante pour nos besoins.

`make` utilise un fichier de configuration nommé `Makefile` (sans extension). Ce fichier indique ce que doit faire `make` lorsqu'il est appelé.

Le `Makefile` est organisé de la façon suivante : il contient une liste de règles avec une *cible*, qui sont des chaînes de caractères suivies de ' :'. Ces cibles sont les arguments que l'on pourra transmettre à `make` pour obtenir un comportement particulier (une cible peut aussi faire référence à une autre cible). Après les ' :' figurent des fichiers ou autres cibles qui sont des *dépendances*, puis, sur la ligne suivante et après une tabulation, figure éventuellement une commande à réaliser pour construire la cible. Il peut même y avoir plusieurs instructions, chacune sur une ligne commençant par une tabulation (attention à ne pas oublier cette tabulation et à ne pas la remplacer par des espaces). Les dépendances indiquent dans quel cas la cible doit être reconstruite.

Une cible peut être un nom de fichier ou non. Si c'est un nom de fichier, les dépendances indiquent dans quel cas de figure la règle doit être appliquée pour reconstruire la cible. Les instructions de la règle seront exécutées si un des fichiers spécifiés en dépendance est plus récent que la cible (ce qui signifie qu'il y a eu des modifications récente et que la cible n'est plus à jour). Si une dépendance est une cible, `make` va déterminer si la règle doit être appliquée et si c'est le cas, cela déclenchera aussi la reconstruction de la cible dont elle dépend.

Exemple de fichier `Makefile` pour compiler l'exécutable `mon_programme` dont les fichiers sources ont été présentés précédemment :

```
1 all: mon_programme
2
3 mes_fonctions.o: mes_fonctions.h mes_fonctions.c
4       gcc -c -o mes_fonctions.o mes_fonctions.c
5
6 mon_programme.o: mon_programme.c mes_fonctions.h
7      gcc -c -o mon_programme.o mon_programme.c
8
```

```
9  mon_programme: mon_programme.o mes_fonctions.o
10      gcc -o mon_programme mon_programme.o \
11         mes_fonctions.o
12
13 clean   :
14      rm -f *.o mon_programme
```

'\' permet de continuer sur la ligne suivante.

Ce `Makefile` contient cinq règles. La première a pour cible `all`. C'est une convention que de commencer par celle-ci. Par défaut, lorsque make est appelé sans argument, c'est la première règle qui est appelée. Elle ne contient aucune instruction mais dispose d'une dépendance vers la règle `mon_programme`. Cette règle se contente donc de faire appel à la règle `mon_programme`. Par convention, cette règle `all` contient en dépendance toutes les règles importantes du `Makefile` (il n'y en a qu'une dans cet exemple, mais il peut en exister plusieurs). Ainsi faire un `make all` permet de construire toutes les cibles importantes. Cette première règle étant exécutée si on appelle `make` sans argument, les trois appels suivants à `make` ont le même résultat, à savoir faire appel à la règle `mon_programme` :

```
make
make all
make mon_programme
```

Que se passe-t-il si on exécute l'une de ces commandes ?

La règle `mon_programme` a deux dépendances, `mon_programme.o` et `mes_fonctions.o`. Lorsque cette règle est appelée, `make` va faire appel aux règles correspondantes (puisque des règles de ce nom existent). La règle `mon_programme.o` a ainsi pour dépendances `mon_programme.c` et `mes_fonctions.h`. Il n'existe pas de règles ayant des cibles de ce nom. `make` va donc se contenter de vérifier s'il existe des fichiers de ce nom et s'ils sont plus récents ou non que `mon_programme.o`. Si c'est le cas, il fait appel à l'instruction spécifiée, à savoir :

```
gcc -c -o mon_programme.o mon_programme.c
```

ce qui permet de créer un fichier `mon_programme.o` à jour.

La règle `mes_fonctions.o` fonctionne de la même façon.

Si une de ces deux cibles a été reconstruite, l'instruction associée à la cible `mon_programme` est appelée, à savoir :

```
gcc -o mon_programme mon_programme.o mes_fonctions.o
```

L'exécutable est donc à présent à jour. Si on refait appel à la même commande, rien ne se passe car les fichiers sources sont plus anciens que les fichiers objets et que l'exécutable.

Avec l'aide de ce `Makefile`, pour compiler `mon_programme`, il suffit donc de taper une des trois commandes listées ci-dessus :

```
make
make all
make mon_programme
```

Chaque fois qu'un fichier source du programme est modifié, il faudra refaire appel à cette commande. Elle ne compilera que le nécessaire et génèrera une version à jour de l'exécutable.

5 POINTEURS

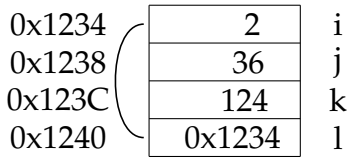
Ce chapitre présente les pointeurs. Cette notion est fondamentale en C et elle est omniprésente. Elle a en fait déjà été utilisée lors des chapitres précédents sans la mentionner explicitement. Après une présentation du principe des pointeurs, les différentes utilisations seront détaillées, en revenant sur les passages des chapitres précédents où il était demandé d'apprendre sans comprendre d'où pouvait venir la syntaxe bizarre qui était utilisée.

5.1 QU'EST-CE QU'UN POINTEUR ?

Les variables utilisées par un programme sont stockées dans la mémoire de l'ordinateur (dans la mémoire vive ou RAM). Cette mémoire est accordée au programme lors de son lancement par le système d'exploitation selon les besoins du programme et elle peut être augmentée ou diminuée pendant l'exécution. Cette mémoire peut être vue comme une succession de cases de 1 octet (8 bits). Chacune de ces cases dispose de sa propre adresse, qui est un entier permettant de la retrouver de façon unique. À partir de cette adresse, il est donc possible de lire la valeur contenue dans la case ou même de la modifier. Un pointeur est une variable qui contient une telle adresse et permet de manipuler la mémoire.

Exemple :

```
1 int i=2,j=36,k=124;
2 int *l=&i;
```



Dans cet exemple, trois variables de type `int` sont déclarées. Chacune occupe quatre octets en mémoire [1]. Le pointeur sur `int l` est déclaré ensuite et initialisé avec l'adresse de la variable `i` (la syntaxe sera détaillée plus tard). Il pointe alors sur la variable `i`.

1. l'adresse `0x1234` est une adresse quelconque, choisie pour l'exemple et qui n'a aucune signification particulière.

Remarque : par convention, il est d'usage de noter les adresses dans la base hexadécimale (base 16), ce que l'on indique en faisant précéder l'adresse des caractères `0x`. `0x1234`, `0xAB3F` ou encore `0x3CD1` sont donc des exemples d'adresse.

5.2 DÉCLARATION ET UTILISATION

Pour déclarer un pointeur, il faut indiquer le type de la variable sur laquelle il pointe, puis un caractère '*', puis le nom du pointeur.

Exemple :

```
1 int i=2;
2 int *l;
```

Cet exemple correspond à la déclaration d'une variable `i` de type `int` et d'un pointeur sur `int` nommé `l`.

Pour récupérer l'adresse d'une variable, il suffit d'ajouter un '&' avant le nom de la variable.

Exemple :

```
3 l=&i;
```

Dans cet exemple, l'adresse de la variable `i` est affectée à `l`.

Pour accéder à la valeur pointée par un pointeur, il faut faire précéder le nom du pointeur du caractère '*'. On parle alors de "déréférencement".

Exemple :

```
4 printf("Valeur␣de␣*l:␣%d\n",*l);
5 *l=3;
6 printf("Valeur␣de␣i:␣%d\n",i);
7 i=4;
8 printf("Valeur␣de␣*l:␣%d\n",*l);
```

Dans cet exemple, la valeur pointée par `l`, autrement dit la valeur de la variable `i`, est affichée, puis cette valeur est modifiée pour prendre la valeur 3. La valeur de `i` est ensuite affichée : `i` a été modifiée par l'instruction précédente, donc la valeur affichée est 3. `i` prend ensuite la valeur 4. Enfin la valeur pointée par `l` est affichée, à savoir 4. Ces instructions provoquent donc l'affichage suivant :

```
Valeur de *l: 2
Valeur de i: 3
Valeur de *l: 4
```

Comme le suggère l'exemple, après l'instruction `l=&i;`, `*l` peut être utilisé de manière équivalente à `i`. Il est ainsi possible de faire avec `*l` tout ce qui était faisable avec `i`. Par exemple :

```
9  (*l)++;
10 *l *= 3;
```

Ces instructions sont donc équivalentes à :

```
9  i++;
10 i *= 3;
```

La première instruction incrémente `i` de 1, puis l'instruction suivante multiplie le résultat par 3. `i` valant initialement 4, il vaut donc 15 après l'exécution de ces instructions.

Un pointeur est une variable contenant une adresse. À ce titre, un pointeur occupe un espace mémoire qui peut être retrouvé à partir de son adresse. Il est donc possible de définir des pointeurs sur des pointeurs... Exemple :

```
1  int n=3;
2
3  int *p=&n;
4
5  int **q=&p;
```

Après ces instructions, `q` pointe sur `p` qui lui-même pointe sur `n`. `*q` correspond à `p` et `**q` correspond à `n`.

5.3 LE POINTEUR NULL

Comme toute variable, dès qu'un pointeur est déclaré, il a une valeur particulière. S'il n'est pas initialisé, comme pour toute autre variable, cette valeur peut être quelconque.

`NULL` est une valeur spéciale, équivalente à la valeur `0`. Elle est utilisée par convention pour indiquer qu'un pointeur n'a pas été initialisé. Il est est nécessaire pour cela d'**initialiser systématiquement un pointeur à la valeur NULL** si on ne sait pas tout de suite quelle valeur lui attribuer : c'est la seule valeur numérique particulière connue.

```
1  int *l = NULL;
2
3  /* instructions pendant lesquelles
4    l peut avoir ete initialise ... ou pas */
5
6  /* pour le verifier: */
7  if (l != NULL) {
8     printf("La␣valeur␣stockee␣a␣l'adresse␣pointee␣");
9     printf("par␣l␣est:␣%d\n",*l);
10 }
```

```
11 else {
12    printf("Attention, l n'a pas ete initialise\n");
13 }
```

5.4 PASSAGE DE PARAMÈTRES DE FONCTION PAR POINTEUR

Il a été vu précédemment que les arguments à une fonction sont transmis par valeur.

Si une fonction prend un pointeur en argument, sa valeur sera copiée, comme pour les autres types d'arguments. Cependant cette valeur est l'adresse d'un bloc mémoire qui, lui, n'est pas recopié et reste unique. Ainsi toute modification qui sera apportée sur le bloc mémoire en question sera gardée à l'issue de la fonction.

Exemple :

```
1  void f(int *n) {
2    *n=*n+1;
3  }
4
5  int main(void) {
6    int i=2;
7    printf("1: i=%d",i);
8    f(&i);
9    printf("2: i=%d",i);
10 }
```

L'exécution de ce programme affichera :

```
1        1: i=2
2        2: i=3
```

Par contre, si le pointeur est lui-même modifié, ces modifications seront perdues. Ainsi, si dans la fonction, `n` est modifié à la place de `*n`, cela n'aura aucun impact en dehors de la fonction (`i` sera inchangé).

5.5 POINTEURS, TYPES ET TABLEAUX

Une adresse mémoire indique l'emplacement d'un seul octet. Or la plupart des variables occupent plus d'un octet. Associer un type à un pointeur permet donc au compilateur de connaître la taille de la zone pointée. Ainsi il existe des pointeurs sur des `int`, des pointeurs sur des `double`, etc. et un

pointeur sur un `int`, pointe donc sur une zone mémoire de 4 octets (sur une architecture de processeur de 32 ou 64 bits) et un pointeur sur un `double` pointe sur une zone mémoire de 8 octets.

La connaissance de la taille de la zone mémoire pointée est indispensable dès lors qu'il s'agit de déréférencer le pointeur, autrement dit d'aller chercher la valeur stockée à l'adresse pointée ou de la modifier.

Cette connaissance autorise également la définition d'opérations arithmétiques adaptées. Il est ainsi possible d'ajouter ou de soustraire un entier à un pointeur. Si on ajoute `n` à un pointeur sur `int l`, l'adresse contenue dans `l` est décalée non pas de `n` octets, mais de `n*sizeof(int)`. Ainsi, quel que soit le type d'un pointeur `p`, lui ajouter 1 le fait pointer sur la variable suivant immédiatement la variable pointée par `p`, lui soustraire 1 le fait pointer sur la variable précédente.

ATTENTION : il est possible d'ajouter ou de soustraire n'importe quel entier à un pointeur, mais cela ne signifie pas pour autant que l'adresse mémoire obtenue pointe sur une zone mémoire autorisée, cette question sera évoquée à la section 5.8.

À quoi de tels calculs arithmétiques peuvent-ils servir ? Ils ont été utilisés pour implanter les tableaux en C. Un tableau de `n int` est une zone mémoire de `n*sizeof(int)` octets consécutifs. Ainsi, si l'on dispose d'un pointeur sur la première "case" du tableau, en lui ajoutant 1 on obtient l'adresse de la deuxième case du tableau, en ajoutant encore 1, on obtient l'adresse de la troisième case du tableau, etc. D'un point de vue syntaxique, il sera équivalent d'utiliser l'écriture tableau avec des '[]' ou l'écriture pointeur avec '*' : l'écriture `t[i]` est équivalente à `*(t+i)` : un tableau est équivalent à un pointeur sur le premier élément du tableau. Exemple :

```
1 int t[10] = {0,1,2,3,4,5,6,7,8,9};
2 int *l;
3 l  = t;
```

Après l'affectation `l=t`, `l` et `t` peuvent être utilisés l'un à la place de l'autre de manière équivalente. Exemples :

```
1 t[0]=10;
2 *(t+1)=11;
3 l[2] = 12;
4 *(l+3) = 13;
```

Remarque : l'instruction `l=t` aurait pu être remplacée par `l=&t[0]` (adresse de la première case du tableau).

Lorsqu'un tableau est passé en argument à une fonction, la taille du tableau est ignorée par le compilateur. En pratique, déclarer qu'un argument est un tableau ou que c'est un pointeur revient au même. La taille indiquée pour le tableau ne peut donc servir qu'aux personnes qui vont lire le code. Si la fonction est appelée avec un tableau de taille incorrecte, aucune erreur

ne sera indiquée lors de la compilation, par contre, selon les cas, une erreur pourra se produire à l'exécution (voir section 5.8).

5.6 POINTEUR GÉNÉRIQUE

Il est parfois nécessaire de manipuler des adresses mémoires sans connaitre le type des données stockées, nous en verrons un exemple dans la section suivante. Dans ce cas, il existe un type de pointeur particulier, le type générique : `void *`. Un pointeur de ce type ne peut être déréférencé et il n'est pas possible de lui ajouter ou soustraire un entier. Il est possible, par contre, de transformer un pointeur générique en pointeur typé et vice-versa.

Exemple :

```
1 int n=3;
2 int *p=&n;
3
4 void *v=p;
5
6 int *q=v;
7
8 *q=4;
```

La dernière instruction affecte la valeur 4 à la variable `n`.

5.7 ALLOCATION DYNAMIQUE

5.7.1 malloc

Jusqu'à présent, les adresses mémoire qui ont été manipulées étaient celles de variables déclarées précédemment ou de tableaux. Il est possible de demander explicitement de la mémoire pour y stocker quelque chose ultérieurement. Pour cela, il faut faire appel à la fonction `malloc`. Son prototype est le suivant :

```
1 void *malloc(size_t taille);
```

L'argument est un entier spécifiant le nombre d'octets souhaités. La fonction renvoie un pointeur sur le premier octet du bloc occupant `taille` octets (ou la valeur `NULL` si l'allocation n'a pas fonctionné). La valeur de retour est de type pointeur générique, `malloc` réservant la mémoire sans savoir ce qu'elle doit contenir.

Exemple : allocation d'un tableau dont la taille est une variable

```
1 int n=3;
```

```
2
3 int *t = (int *) malloc(n*sizeof(int));
```

Le résultat du `malloc` est transformé en un `int *`. Cette instruction permet d'allouer un tableau de `n` entiers. Le nombre d'octets nécessaire est donc de `n` fois la taille d'un `int` en octets, autrement dit `n*sizeof(int)`.

Remarque : la fonction `strdup` est équivalente à l'allocation d'un tableau de caractères puis à une copie :

```
1 char *from="chaine␣initiale";
2 char *to;
3
4 to=strdup(from);
5
6 /* equivalent a: */
7
8 to=malloc(sizeof(char)*(len(from)+1));
9 strcpy(to,from);
```

5.7.2 free

Tout bloc mémoire alloué avec un `malloc` doit être libéré par un `free`.

```
1 void free(void *);
```

Exemple : libération du tableau alloué précédemment

```
1 free(t);
```

Pourquoi allouer explicitement de la mémoire ? Il peut être nécessaire de créer un tableau dont la taille n'est pas connue à l'avance, or lors de la déclaration d'un tableau, il faut spécifier sa taille sous la forme d'une constante [2]. L'allocation dynamique permet d'allouer de l'espace pour un tableau dont la taille n'est pas une constante connue à l'avance.

Une deuxième raison est liée à la portée d'une variable. Une variable déclarée dans une fonction n'existe qu'à l'intérieur de cette fonction. À la fin de la fonction, la mémoire qui avait été allouée (automatiquement) à la variable est libérée [3]. Il est parfois nécessaire de disposer de variables ou de tableaux qui "survivent" à la fonction où ils ont été déclarés. Lorsqu'un bloc mémoire est alloué dynamiquement, il ne sera pas libéré automatiquement,

2. Il existe une nouvelle norme du langage C, la norme C99 qui relâche cette contrainte, mais les normes les plus anciennes, comme les normes ANSI-C89 et ISO-C90 sur lesquelles se base ce cours, ne l'autorisent pas.

3. sauf dans le cas d'une variable `static`.

il faut le libérer explicitement avec un `free`. Le bloc mémoire peut donc "survivre" à la fonction dans laquelle il a été alloué.

Exemple : définition d'une fonction permettant d'allouer un tableau dont la taille est donnée en argument (et exemple d'utilisation) :

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  unsigned int *allouerTableau(int taille) {
5     /* alloue un tableau de taille 'taille'
6     et initialise ses cases a la valeur 0 */
7     unsigned int i;
8     unsigned int *tab=(unsigned int *)malloc(
9          taille*sizeof(unsigned int));
10    if (tab==NULL) {
11       printf("Probleme␣d'allocation\n");
12       return NULL;
13    }
14    for (i=0;i<taille;i++) {
15       tab[i]=0;
16       /* ou de facon equivalente:
17        *(tab+i)=0; */
18    }
19    return tab;
20 }
21
22 int main() {
23
24   unsigned int s,i;
25   unsigned int *p;
26
27  printf("Saisir␣la␣taille␣du␣tableau:␣");
28  scanf("␣%d",&s);
29
30  p=allouerTableau(s);
31
32  /* on utilise le tableau */
33  p[0]=1;
34  for (i=1;i<s;i++) {
35    p[i]=p[i-1]*i;
36  }
37  for (i=0;i<s;i++) {
38    printf("p[%u]=%d\n",i,p[i]);
39  }
40
```

```
41  /* on libere le tableau */
42  free(p);
43
44  return 1;
45 }
```

5.7.3 realloc

`malloc` permet d'allouer un bloc de mémoire. Pour changer sa taille, il faudrait allouer un autre bloc de mémoire de la bonne taille, puis copier les octets communs.

L'exemple suivant reprend la fonction d'allocation d'un tableau vue ci-dessus. La fonction définie a pour but de changer la taille du tableau en utilisant un `malloc` :

```
1  int *changerTailleTableau(int *tab,
2          int ancienne_taille, int nouvelle_taille) {
3     /* ATTENTION, la version avec realloc
4          doit etre preferee car plus simple
5          et plus efficace */
6     unsigned int i, taille;
7     int *ntab=(int *)malloc(nouvelle_taille*sizeof(int));
8     if (ntab==NULL) {
9        printf("Probleme d'allocation\n");
10       return NULL;
11    }
12
13    /* on recupere dans taille le minimum entre
14    ancienne_taille et nouvelle_taille   */
15    taille = ancienne_taille>nouvelle_taille?
16             nouvelle_taille:ancienne_taille;
17
18    for (i=0;i<taille;i++) {
19       ntab[i]=tab[i];
20       // ou de facon equivalente:
21       // *(ntab+i)=*(tab+i);
22    }
23
24    free(tab);
25
26    return ntab;
27 }
28
29 int main() {
```

```
30
31   unsigned int s1, s2,i;
32   int *p1,*p2;
33
34  printf("Saisir␣la␣taille␣du␣tableau:␣");
35  scanf("␣%d",&s1);
36
37  p1=allouerTableau(s1);
38
39  for (i=0;i<s1;i++) {
40    p1[i]=i;
41  }
42
43  printf("Saisir␣la␣nouvelle␣taille␣du␣tableau:␣");
44  scanf("␣%d",&s2);
45
46  p2=changerTailleTableau(p1,s1,s2);
47
48  free(p2);
49
50  return 1;
51 }
```

Il est possible de réaliser les mêmes opérations avec une seule instruction :
`void *realloc(void *adr, size_t taille)`

La fonction `realloc` change la taille d'un emplacement mémoire précédemment alloué. La nouvelle taille est `taille`, elle peut être inférieure ou supérieure. Les octets conservés de l'ancien tableau sont identiques et l'emplacement mémoire peut changer. Dans ce cas, les valeurs communes entre l'ancien tableau et le nouveau sont automatiquement copiées dans le nouveau tableau. Cette fonction doit être préférée à l'utilisation d'un `malloc` et de copies car, si c'est possible, elle change directement la taille du bloc mémoire alloué et évite ainsi des copies.

La fonction `changerTailleTableau` peut être définie de la façon suivante avec le `realloc` :

```
1 /* Il est inutile de connaitre l'ancienne taille
2 du tableau */
3 int *changerTailleTableau2(int *tab,
4          int nouvelle_taille) {
5    int *ntab=(int *)realloc(tab,
6                             nouvelle_taille*sizeof(int));
7    if (ntab==NULL) {
8       printf("Probleme␣d'allocation\n");
9       return tab;
```

```
10      }
11
12      return ntab;
13  }
```

Remarquez qu'il n'est pas nécessaire de libérer `tab`. Si l'appel à `realloc` s'est bien passé, `tab` est libéré automatiquement, si besoin est. Dans le cas contraire, `tab` n'est pas libéré, pour que les données soient toujours accessibles.

5.8 ERREUR DE SEGMENTATION

Pendant son exécution, un programme occupe un espace limité dans la mémoire vive de l'ordinateur tant pour stocker ses propres instructions (le code compilé qui ext chargé en mémoire au lancement du programme) que pour stocker ses données. Une valeur choisie au hasard et utilisée comme adresse mémoire ne tombera probablement pas dans une zone à laquelle le programme a accès. Dans ce cas, le système d'exploitation l'indiquera violemment au travers d'une erreur faisant "planter" le programme (en général, il s'agira d'une erreur dite de segmentation "seg fault" en anglais, mais elle peut prendre d'autres noms comme "bus error", par exemple). Chaque fois que des erreurs de ce type apparaîtront, cela signifiera qu'il y a eu une mauvaise manipulation de la mémoire. Des outils comme `ddd` ou `valgrind` peuvent aider à trouver les instructions incriminées.

En cas d'erreur de ce type, il faut pister toutes les utilisations de pointeurs (ou de tableaux). Les erreurs les plus courantes sont les suivantes :

— utilisation d'un pointeur non initialisé. Il contient une valeur quelconque et pointera donc a priori en dehors de la zone mémoire autorisée. Il est à noter que s'il pointe dans la zone mémoire autorisée, cela peut provoquer des erreurs très difficiles à détecter ;
— déréférencement d'un pointeur valant NULL ;
— débordement de tableau : si `t` est un tableau de 10 `int`, `t[10]=42` correspond à la modification des 4 octets qui suivent immédiatement le tableau. Cette instruction est donc susceptible de créer une erreur de segmentation. D'une manière générale `t[n]=42` est susceptible de provoquer une erreur de segmentation pour toute valeur de `n` hors de l'intervalle 0 - 9.

Il est à noter que l'on ne donnera jamais directement une valeur numérique particulière à un pointeur (une seule exception à cette règle : le cas de la valeur `NULL`).

6 STRUCTURES

6.1 DÉFINITION D'UNE STRUCTURE

Les structures sont des types composés définis par le programmeur. Les structures regroupent à l'intérieur d'une même variable des variables **sémantiquement liées**. Exemple : une fiche d'un répertoire contient le nom, le prénom, l'adresse, le numéro de téléphone...

Exemple simple : point dans un espace 3D

```
1 struct point {
2   float x;
3   float y;
4   float z;
5 };
6 struct point p1;
7 p1.x = 12.0;
8 p1.y = 24.2;
9 p1.z = 42.0;
```

Une structure est déclarée de la façon suivante :

```
1 struct <nom de type>
2 {
3    type1 var1;
4    type2 var2;
5    ...
6    typen varn;
7 };
```

Pour déclarer une variable du type ainsi défini, il faut répéter le mot-clé `struct`, puis le nom choisi pour la structure et enfin le nom de la variable : `struct <nom de type> <nom de variable>`.

Exemple : `struct point p1;`

L'accès aux différents champs d'une structure se fait en indiquant le nom de la variable, puis un caractère '.', puis le nom du champ : `var.<nom du champ>`.

Exemple :`p1.x=44.20;`

6.2 TYPEDEF

Les structures définies comme indiqué dans la section précédente nécessitent de taper systématiquement le mot-clé `struct` chaque fois que l'on souhaite déclarer une variable, ce qui n'est pas très pratique. Le langage C offre un mécanisme pour éviter cela : le mécanisme de définition de types synonymes. Ce mécanisme permet de définir un nouveau type que l'on peut utiliser de façon équivalente au type original. Pour définir ce type synonyme, il suffit de donner le mot-clé `typedef`, puis le type initial, puis le nom du type synonyme.

Exemple :

```
1 typedef int entier;
2 entier i=3;
```

Exemple avec un tableau :

```
1 typedef int vect[3];
2 vect v1,v2;
3 v1[0]=3.2;
```

Il est possible de définir des types synonymes à des structures. Cela permet de remplacer le nom de type `struct <nom de type>` par le nom de type synonyme seul.

Exemple :

```
1  struct point {
2    float x;
3    float y;
4    float z;
5  };
6  typedef struct point spoint;
7
8  spoint p;
9  p.x=1.2;
10 p.y=3.4;
11 p.z=5.6;
```

Il est à noter que la déclaration du type comme l'association avec un nom de type synonyme peut se faire en une seule opération. Dans ce cas, le mot-clé `typedef` est suivi par le nom de structure (`struct <nom de type>`), puis par sa déclaration avant de finir par le nom du type synonyme.

Exemple :

```
1 typedef struct point {
2   float x;
3   float y;
4   float z;
```

```
5 } spoint;
6
7 spoint p;
```

6.3 UTILISATION DES STRUCTURES

Une structure peut être initialisée comme un tableau, en spécifiant entre '{}' les valeurs des différents champs.

Exemple 1 :

```
1 spoint a={1.0, 2.0, 3.0};
```

Exemple 2 :

```
1 typedef struct _ma_struct {
2   float x;
3   char c;
4   int i;
5 } ma_struct;
6
7 ma_struct ms={1.234, 'f', 3};
```

Il est possible d'affecter la valeur d'une structure à une autre structure. Cela revient à faire une affectation pour chacun des champs.

Exemple :

```
1 spoint a={1.0, 2.0, 3.0}, b;
2
3 b=a;
4
5 /* equivalent a */
6 b.x = a.x;
7 b.y = a.y;
8 b.z = a.z;
```

Une structure peut contenir des champs de type tableau et il est également possible de définir des tableaux de structures. La syntaxe à utiliser consiste à combiner l'écriture avec un '.' et les '[]' du tableau.

Exemple :

```
1 struct date {
2   unsigned char jour;
3   unsigned char mois;
4   unsigned int annee;
5 };
6
```

```
7  struct personne {
8    char nom[30];
9    char prenom[30];
10   struct date date_naissance;
11 };
12
13 struct personne client={"Dupond","Albert",
14   {24,5,1970}};
15
16 if (client.date_naissance.annee>1960) {
17   printf("Le client est ne apres 1960\n");
18 }
19
20 client.nom[5] = 't';
21 printf("Nom du client: %s\n",client.nom);
22 /* affiche:
23 Nom du client: Dupont
24 */
25
26 struct personne tab[10];
27 strcpy(tab[0].nom,"Deckard");
28 strcpy(tab[0].prenom,"Rick");
29
30 strcpy(tab[1].nom,"Tyrell");
31 strcpy(tab[1].prenom,"Eldon");
```

6.4 STRUCTURES ET POINTEURS

Pour déclarer un pointeur sur une variable de type structure, comme pour les autres types, il suffit d'ajouter un caractère '*' avant le nom de la variable au moment de sa déclaration.

Exemple : `struct point *pp1`.

L'accès aux différents champs de la structure depuis un pointeur se fait avec les caractères '->' : `pvar-><nom du champ>`.

Exemple :`pp1->x=44.20;`

Remarque : en déréférençant un pointeur sur une variable de type structure, il est possible de récupérer la valeur d'un champ avec un '.'.

Exemple :`(*pp1).x=44.20;`

Une structure peut également contenir des champs qui sont eux-mêmes des pointeurs.

Exemple :

```
1 typedef struct _personne {
```

```
2   char * nom;
3   char * prenom;
4 } personne;
5
6
7 personne p1,p2;
8
9 p1.nom = (char *)malloc(7+1);
10 strcpy(p1.nom,"Deckard");
11
12 p1.prenom = strdup("Rick");
13 // strdup equivalent a malloc puis strcpy
14
15 p2=p1; // cela fait quoi ?
16
17 printf("p2.nom=%s␣p2.prenom=%s\n",p2.nom,p2.prenom);
18 free(p1.nom);
19 free(p1.prenom);
20
21 // Quel est le resultat de cet appel ?
22 printf("p2.nom=%s␣p2.prenom=%s\n",p2.nom,p2.prenom);
```

Le premier `printf` affiche :

```
p2.nom=Deckard p2.prenom=Rick
```

Le second `printf` provoque une erreur à cause du free, qui a libéré la mémoire qui n'est plus accessible désormais.

Une affectation entre variables d'un type structure contenant un champ de type pointeur recopie la valeur du pointeur. Après l'affectation les deux champs pointent donc sur la même donnée.

Autre exemple :

```
1  typedef struct _mon_tableau {
2    float *tab;
3    int taille;
4  } mon_tableau;
5
6  mon_tableau t1,t2;
7
8  t1.taille=10;
9  t1.tab = (float *)malloc(10*sizeof(float));
10
11 t2=t1;
12
13 t2.tab[0]=1.23; /* modifie egalement t1.tab[0] */
```

```
14 printf("%f\n",t1.tab[0]);
15
16 t1.tab[1]=4.56; /* modifie egalement t2.tab[1] */
17 printf("%f\n",t2.tab[1]);
18
19 printf("t1.tab=%p␣t2.tab=%p\n",t1.tab, t2.tab);
20 /* le code %p permet d'afficher un pointeur
21         en hexadecimal */
```

Ces instructions affichent :

```
1.230000
4.560000
t1.tab=0x100100080 t2.tab=0x100100080
```

Après l'affectation, les champs `tab` de `t1` et `t2` contiennent la même adresse et pointent donc sur la même zone mémoire. Cette zone mémoire devra en particulier n'être libérée qu'une seule fois, soit avec un `free(t1.tab)`, soit, de façon équivalente, avec un `free(t2.tab)`.

6.5 STRUCTURE CONTENANT DES STRUCTURES

Une structure peut contenir des champs qui sont eux-mêmes des structures. L'accès aux champs de la structure se fait alors en enchaînant les '.' dans le cas d'une variable et '->' dans le cas d'un pointeur.

Exemple :

```
1     typedef struct _point2D {
2        float x;
3        float y;
4     } point2D;
5
6     /* le rectangle est defini par deux points
7       (point en bas a gauche et point en haut a droite)
8     */
9     typedef struct _rectangle{
10       point2D bg;
11       point2D hd;
12    } rectangle;
13
14    rectangle r1;
15    r1.bg.x=0;
16    r1.bg.y=0;
17
18    r1.hd.x=10;
```

```
19      r1.hd.y=5;
20
21     /* r1 est le rectangle defini par les points (0,0)
22      et (10,5) */
23
24     /* la meme structure, mais avec des pointeurs */
25      typedef struct _rectanglebis{
26         point2D *bg;
27         point2D *hd;
28      } rectanglebis;
29
30     point2D p1={0,0};
31     point2D p2={10,5};
32
33     rectanglebis r2;
34     r2.bg=&p1; /* adresse de p1 */
35     r2.hd=&p2; /* adresse de p2 */
36
37     printf("Coordonnees␣de␣r2:␣\n");
38     printf("␣␣␣␣bg=(%f,%f)",r2.bg->x,r2.bg->y);
39     printf("␣␣␣␣hd=(%f,%f)",r2.hd->x,r2.hd->y);
```

Remarquez l'écriture `r2.bg->x` qui permet de récupérer le champ `bg` de `r2` (à l'aide d'un '.', `r2` étant une variable) puis le champ `x` de `r2.bg` (à l'aide d'une '->' car le champ `bg` est un pointeur). Pour mémoire, ce qui détermine s'il faut utiliser un '.' ou une '->', c'est le type de ce qui est indiqué du côté gauche : c'est une '->' s'il s'agit d'un pointeur, un '.' sinon.

6.6 LECTURE/ÉCRITURE BINAIRE

L'écriture formatée a été vue précédemment. Elle consiste à transformer tout ce que l'on souhaite écrire en caractères et à les ajouter dans un fichier ou bien à lire des caractères que l'on va ensuite retraduire sous une autre forme. Ainsi, si l'on veut écrire l'entier 42, une fonction d'écriture formatée comme `fprintf` transforme cette valeur en la chaîne de caractères "42" qui est ensuite écrite dans le fichier (il faudra ajouter un caractère séparateur, par exemple un espace ou un retour à la ligne avant d'écrire une autre donnée). La lecture formatée est le symétrique, on lit dans le fichier des caractères que l'on va transformer ensuite dans le type désiré à l'aide d'un `fscanf` (ou d'un `fgets` et d'un `sscanf`).

La lecture écriture binaire procède différemment. Elle recopie directement, octet par octet, le contenu de la mémoire sur le disque dur, sans transformation intermédiaire en caractères. Cela a plusieurs avantages : on peut écrire

directement, avec une seule instruction, un bloc mémoire contenant un tableau ou une structure (ou un tableau de structures...) et le relire tout aussi facilement avec une seule instruction (c'est la raison pour laquelle cette notion est présentée dans ce chapitre). Ces fonctions manipulant la mémoire, elles utilisent des pointeurs, c'est la raison pour laquelle elles n'ont pas été présentées dans le chapitre sur les entrées/sorties, chapitre qui précédait celui sur les pointeurs.

L'inconvénient de cette approche est que le fichier que l'on a ainsi créé est moins *portable* que dans le cas de la lecture/écriture formatée. En effet, comme il a été vu dans le chapitre de rappel, tous les types n'occupent pas le même nombre d'octets selon l'architecture du processeur (16, 32 ou 64 bits). De plus, les nombres ne sont pas toujours rangés dans le même ordre en mémoire, parfois les premiers octets sont ceux de poids faible (architecture petit boutien ou little endian en anglais), parfois ce sont les octets de poids fort qui sont rangés en premiers (architecture grand boutien ou big endian). Étudier ces différentes architectures va au-delà des objectifs de ce cours, néanmoins il est important d'être conscient de ces différences car la conséquence de cette variabilité est qu'une donnée écrite en binaire ne peut être relue correctement que sur un processeur d'architecture similaire.

Pour lire ou écrire dans un fichier en mode binaire, il faut ajouter un caractère "b" au mode d'ouverture.

Exemples :

```
1  /* ouverture en mode lecture binaire */
2 FILE *frb = fopen("fichier1.txt","rb");
3
4  /* ouverture en mode ecriture binaire */
5 FILE *fwb = fopen("fichier2.txt","wb");
```

6.6.1 Écriture binaire

La fonction d'écriture binaire est la suivante :

```
1 size_t fwrite (const void *ptr, size_t size,
2                size_t nmemb, FILE *stream);
```

La fonction `fwrite` écrit `nmemb` éléments de données stockés à l'adresse indiquée par `ptr`, chacun d'eux représentant `size` octets de long, dans le flux pointé par `stream`. Cette fonction écrit donc au plus les `nmemb` × `size` octets stockés à l'adresse pointée par `ptr` dans le fichier. La valeur renvoyée est égale à `nmemb` si tout s'est bien passé, ou au nombre d'éléments qui ont pu être écrits si l'écriture n'a pas pu aboutir complètement (par exemple s'il ne restait pas assez de place pour tout écrire sur le disque dur).

Exemple :

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

int main() {
  int tab[10]={1,2,3,4,5,6,7,8,9,10};

  /*  ouverture du fichier */
  FILE *fichier=fopen("fichier_tab","wb");
  if (fichier==NULL) {
    printf("Erreur lors de l'ouverture du fichier.");
    exit(1);
  }

  /* ATTENTION: pas de '&' car tab est un tableau
   ... c'est donc aussi un pointeur sur son premier
   element ! */
  fwrite(tab,sizeof(int),10,fichier);

  /* fermeture du fichier */
  fclose(fichier);

  return 0;
}
```

Remarque : nous n'avons pas mis l'extension ".txt" au fichier de destination car il ne s'agit pas d'un fichier texte habituel, mais d'un fichier binaire. Vous pouvez essayer de l'ouvrir avec un éditeur de texte, vous verrez apparaître des caractères étranges.

6.6.2 Lecture binaire

La fonction de lecture binaire est la suivante :

```c
size_t fread (void *ptr, size_t size,
              size_t nmemb, FILE *stream);
```

La fonction `fread` lit `nmemb` éléments de données, chacun d'eux représentant `size` octets de long. Elle les lit depuis le flux pointé par `stream`, et les stocke à l'adresse contenue dans `ptr`. Comme précédemment, la valeur de retour indique ce qui a pu être lu (au maximum `nmemb` valeurs, mais éventuellement moins si la fin du fichier a été rencontrée avant).

L'exemple suivant permet de lire les données écrites dans l'exemple précédent :

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

```c
#include <stdlib.h>

int main() {

  int i;
  int tab[10];

  /*  ouverture du fichier */
  FILE *fichier=fopen("fichier_tab","rb");
  if (fichier==NULL) {
    printf("Erreur␣lors␣de␣l'ouverture␣du␣fichier.");
    exit(1);
  }

  /* ATTENTION: pas de '&' car tab est un tableau ! */
  fread(tab,sizeof(int),10,fichier);

  for (i=0;i<10;i++) {
    printf("tab[%d]=%d\n",i,tab[i]);
  }

  /* fermeture du fichier */
  fclose(fichier);

  return 0;
}
```

6.7 ENTRÉES/SORTIES ET STRUCTURES

Les fonctions d'entrées/sorties binaires vues à la section précédente permettent d'écrire ou de lire très facilement des structures. Exemple :

```c
typedef struct _point {
  float x;
  float y;
  float z;
} point;
point p1={ 1.0, 2.0, 3.0},p2;

FILE *f=fopen("mon_fichier","wb");
fwrite(&p1,sizeof(point),1,f);
fclose(f);
f=fopen("mon_fichier","rb");
```

```
12 fread(&p2,sizeof(point),1,f);
```

Dans cet exemple, le `point p1` est écrit dans le fichier "mon_fichier", puis ce fichier est fermé et réouvert en lecture. Le `point p2` est alors lu. Après ces instructions, `p1` et `p2` sont identiques.

Ce qui est très pratique, c'est que l'on peut faire de même si la structure contient des tableaux :

```
1 typedef struct _contact {
2   char nom[30];
3   char prenom[30];
4 } contact;
5 contact c1={"Tyrell", "Eldon" };
6
7 FILE *f=fopen("mon_fichier","wb");
8 fwrite(&c1,sizeof(contact),1,f);
9 fclose(f);
```

ATTENTION : si la structure contient des pointeurs, ce qui est écrit, c'est la valeur du pointeur (autrement dit l'adresse mémoire qu'il contient) et non ce sur quoi elle pointe... Exemple :

```
1
2 typedef struct _contact {
3   char * nom;
4   char * prenom;
5 } contact;
6 contact c1;
7 c1.nom = strdup("Tyrell");
8 c1.prenom = strdup("Eldon");
9
10 FILE *f=fopen("mon_fichier","wb");
11
12 // ATTENTION: ne marche pas !!
13 fwrite(&c1,sizeof(contact),1,f);
14 // cela ecrit les adresses memoire et non le contenu...
15
16 fclose(f);
```

Dans le cas de champs de type pointeurs, il n'y a pas d'autres choix que de les traiter séparément. Exemple :

```
1
2 typedef struct _contact {
3   char * nom;
4   char * prenom;
5 } contact;
```

```
6  contact c1;
7  c1.nom = strdup("Tyrell");
8  c1.prenom = strdup("Eldon");
9
10 FILE *f=fopen("mon_fichier","w");
11 fprintf(f,"%s\n",c1.nom);
12 fprintf(f,"%s\n",c1.prenom);
13 fclose(f);
```

et la fonction de lecture associée :

```
1
2  contact c2;
3
4  FILE *f=fopen("mon_fichier","r");
5  char buffer[128];
6  fgets(buffer,128,f);
7  /* pour enlever le '\n'
8   on remplace le dernier caractere
9   (\n) par \0 */
10 buffer[strlen(buffer)-1]='\0';
11 c2.nom=strdup(buffer);
12
13 fgets(buffer,128,f);
14 /* pour enlever le '\n' */
15 buffer[strlen(buffer)-1]='\0';
16 c2.prenom=strdup(buffer);
17
18 fclose(f);
```

Ces différents exemples utilisent soit la lecture/écriture binaire, soit la lecture/écriture formatée. N'hésitez pas à tester ces programmes et à observer le contenu des fichiers ainsi créés.

7 LISTES CHAÎNÉES

Ce chapitre présente le principe des listes dites chaînées et leur implantation en langage C. Ce type de structure est très courant en informatique et son implantation en C est l'occasion d'utiliser de façon plus approfondie les structures et les pointeurs qui ont été vus auparavant. Il est indispensable d'avoir bien compris ces deux chapitres avant d'aborder celui-ci. Les listes chaînées servent à stocker un certain nombre de variables de même type, de même que les tableaux. Elles sont à utiliser dans certains cas de figures qui seront explicités dans la section suivante, avant de présenter leur implantation en C, puis les fonctions de manipulation de listes chaînées les plus courantes.

7.1 INTRODUCTION

7.1.1 Des limitations des tableaux

Les tableaux permettent de stocker un nombre fixé à l'avance de variables d'un type donné. Par exemple, un tableau déclaré de la façon suivante : `int tab[10]` permet de stocker 10 variables de type `int`, stockées les unes après les autres.

Les tableaux sont très simples à définir et à utiliser, cependant, ils disposent de quelques inconvénients. Tout d'abord, la taille, qui est imposée à la déclaration (ou à l'allocation) peut être un problème. Si on souhaite agrandir un tableau, il faut allouer un tableau de taille supérieure, recopier le contenu de l'ancien tableau dans le nouveau, puis libérer la mémoire allouée à l'ancien tableau. La fonction `realloc` effectue toutes ces opérations (voir section 5.7.3). S'il est possible d'agrandir le tableau, cette fonction le fera, évitant ainsi d'avoir à recopier le contenu du tableau :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main (void) {
6
7   int *tab1=NULL, *tab2=NULL;
8   int i;
9   /* on alloue le tableau */
```

```
10
11    tab1= (int *)malloc(10*sizeof(int));
12    if (tab1==NULL) {
13      printf("Erreur␣lors␣de␣l'allocation\n");
14      exit(1);
15    }
16
17    /* on initialise le tableau */
18    for (i=0;i<10;i++) {
19      tab1[i]=i;
20    }
21
22    /* on souhaite agrandir le tableau:
23     - on alloue plus de memoire
24     - on recopie si besoin en meme temps
25     - on libere la memoire qui doit etre liberee  */
26    tab2=(int*)realloc(tab1,20*sizeof(int));
27    if (tab2==NULL) {
28      printf("Erreur␣lors␣de␣l'allocation\n");
29      exit(1);
30    }
31
32    /* on peut alors continuer a remplir le tableau */
33    for (i=10;i<20;i++) {
34      tab2[i]=i;
35    }
36
37    for (i=0;i<20;i++) {
38      printf("tab2[%d]=%d\n",i,tab2[i]);
39    }
40
41    free (tab2);
42
43    return 0;
44  }
```

Cette fonction n'évite pas la recopie dans le cas général. S'il ne reste pas assez de mémoire libre après le tableau initial, le tableau plus grand que l'on souhaite allouer sera nécessairement ailleurs en mémoire étant donné que toutes les cases du tableau doivent être contiguës. Dans le cas de tableaux de grande taille, cette opération peut être très coûteuse. En tout état de cause, elle aura un coût proportionnel à la taille du tableau initial.

Le cas de l'agrandissement d'un tableau n'est pas le seul nécessitant une recopie massive d'éléments de ce tableau. Supposons que l'on dispose d'un

tableau trié auquel on souhaite ajouter des données. Dans le cas le moins favorable (si la donnée à ajouter se place avant le premier élément du tableau), il faudra recopier TOUT le tableau afin de décaler le contenu pour libérer une case dans laquelle écrire la nouvelle donnée.

Exemple :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define TAILLE 20
6
7  int main (void) {
8
9    int tab[TAILLE]={1,2,3,4,5,6,7,8,9,10};
10
11   /* nombre de valeurs stockees dans le tableau */
12   int nbval=10;
13   int i;
14
15   /* on souhaite ajouter 0 dans le tableau */
16
17   /* il faut commencer par decaler son contenu */
18   for (i=nbval-1;i>=0;i--){
19     tab[i+1]=tab[i];
20   }
21
22   /* puis on peut mettre 0 en place */
23   tab[0]=0;
24
25   /* le tableau contient maintenant
26   un element supplementaire */
27   nbval++;
28
29   for (i=0;i<nbval;i++) {
30     printf("tab[%d]=%d\n",i,tab[i]);
31   }
32
33   return 0;
34 }
```

Autrement dit, les tableaux disposent de limitations qui sont que l'agrandissement d'un tableau de même que l'insertion (ou la suppression) d'un élément sont des opérations qui ont un coût dépendant de la taille du tableau. Dans le cas de tableaux de grande taille, ces opérations peuvent donc

être très coûteuses, or il peut être nécessaire de les réaliser fréquemment. Il faut alors se tourner vers des structures qui permettent de réaliser ces opérations dans un temps constant, indépendant de la taille du tableau (et idéalement très court...).

7.1.2 Idée intuitive

Les contraintes que l'on a identifiées sont liées au fait que toutes les cases d'un tableau sont stockées les unes à la suite des autres. Cela impose de disposer de suffisamment d'espace contigu et implique de déplacer potentiellement de grandes quantités de données si l'on souhaite insérer de nouvelles valeurs dans le tableau. Pourquoi ne pas définir une nouvelle structure dans laquelle les éléments ne se suivent pas les uns après les autres ? Dans le cas d'un tableau, l'élément qui suit une case particulière est obtenu en ajoutant `1` au pointeur sur cette case. Si l'on ne souhaite plus stocker les cases de ce nouveau type de tableau les unes à la suite des autres, il faut se donner un moyen de les retrouver. Pourquoi ne pas indiquer, dans chaque case, l'emplacement de la case suivante ? De cette façon, en partant de la case initiale, on pourra retrouver toutes les cases de ce "tableau". C'est le principe des listes chaînées. Chaque élément contient une donnée et l'adresse de l'élément suivant.

Qu'est-ce que cela implique ? Les différents éléments d'une liste chaînée n'ont pas besoin d'être les uns à la suite des autres. On peut les allouer séparément et stocker l'adresse mémoire d'un élément dans l'élément qui le précède pour pouvoir le retrouver. Lorsque l'on souhaite ajouter un élément à une telle liste, il suffit donc d'allouer un élément et un seul, et de mettre à jour la "chaîne". Pour mettre à jour la chaîne, il suffit de modifier les éléments qui indiquent l'emplacement de l'élément à ajouter, donc a priori uniquement l'élément qui le précède [1]. Ce qu'il faut noter ici, c'est que cette opération a un coût qui ne dépend pas de la taille de la liste. On a donc résolu ce qui nous posait problème.

Quelle conséquence a l'utilisation de listes chaînées ? Si l'on a réussi à résoudre des problèmes liés aux tableaux, il faut se demander si on a introduit d'autres problèmes. Si ce n'était pas le cas, on pourrait en effet remplacer systématiquement les tableaux par des listes chaînées sans se poser plus de questions. Avec les listes chaînées, on introduit en fait deux problèmes potentiels. Le premier est que l'on ne peut plus accéder directement à un élément de la liste. Dans le cas d'un tableau, on sait que pour trouver le 4eme élément d'un tableau `tab`, il suffit d'écrire `tab[3]` ou, écrit avec des pointeurs, `*(tab+3)` (le premier élément d'un tableau est `tab[0]`). L'accès à un élément est donc immédiat et indépendant de la taille du tableau

1. Nous verrons plus tard des listes doublement chaînées qui contiennent à la fois l'adresse de l'élément suivant, et celle de l'élément précédent. Dans ce cas, il faudra aussi modifier l'élément suivant.

(il suffit de faire l'addition d'un entier à un pointeur). Dans le cas des listes chaînées, on ne connait pas à l'avance l'emplacement d'un élément : il est uniquement spécifié dans l'élément qui le précède. Il est donc nécessaire de parcourir la liste depuis son premier élément jusqu'à trouver l'élément cherché. Cette opération n'est plus indépendante du nombre d'éléments stockés dans la liste ! De plus, chaque élément doit contenir l'adresse mémoire de l'élément suivant, ce qui n'était pas utile avec un tableau. Une liste chaînée va donc occuper plus de mémoire qu'un tableau.

Quand choisir d'utiliser une liste chaînée plutôt qu'un tableau ? Nous avons vu que cette structure dispose d'avantages, mais aussi d'inconvénients. Elle sera donc à utiliser dans les cas où il faut insérer ou supprimer fréquemment des éléments. Le choix d'une telle structure ne saurait donc être systématique et nécessite donc de bien réfléchir à l'usage qui en sera fait. Tout ce que l'on peut faire avec une liste chaînée pourrait être fait avec un tableau. Seules des raisons d'efficacité peuvent pousser à utiliser de préférence l'un ou l'autre.

7.2 DÉFINITION D'UNE LISTE CHAÎNÉE EN C

Une liste chaînée est définie par une succession d'éléments contenant une donnée et l'adresse mémoire du prochain élément de la liste. Le type à définir pour caractériser une liste chaînée est donc uniquement le type d'un de ses éléments. Ce type respecte le schéma suivant :

```
1 struct _un_element{
2
3    /* champs de donnees */
4
5    struct _un_element *suivant;
6 };
```

Les données stockées peuvent être de n'importe quel type et peuvent nécessiter autant de champs que nécessaire. Le seul impératif est l'existence d'un champ de type pointeur sur la structure ainsi définie (son nom, de même que le nom choisi pour la structure, importe peu, nous utiliserons le nom `suivant` dans la suite de ce cours). Pour faciliter l'utilisation de cette structure, nous définirons simultanément un type synonyme de la façon suivante :

```
1 typedef struct _un_element{
2
3    /* champ(s) de donnees */
4
5    struct _un_element *suivant;
6 } Un_element;
```

Il est à remarquer que le type `Un_element` ne peut pas être utilisé dans la définition du champs `suivant` car il n'est pas encore connu par le compilateur... Comme nous aurons fréquemment besoin de manipuler des pointeurs sur cette structure, nous définirons souvent un type comme synonyme à un pointeur sur cette structure :

```
1 typedef struct _un_element *P_un_element;
2 typedef struct _un_element{
3
4    /* champ(s) de donnees */
5
6    P_un_element suivant;
7 } Un_element;
```

Notez que lorsque `P_un_element` est défini, `struct _un_element` n'est pas encore connu. Cela ne pose pas de problème au compilateur, qui l'autorise à partir du moment ou l'on définit un type "pointeur sur". Le compilateur n'aurait ainsi pas autorisé la définition du type `Un_element` de cette façon.

Une liste chaînée est ensuite simplement définie comme un pointeur sur un de ces éléments. En pratique, la liste chaînée pointera sur le premier élément qu'elle contient. Une liste chaînée ne nécessite donc pas la définition d'un autre type, il suffit de déclarer un pointeur :

```
1 P_un_element ma_liste=NULL;
```

La variable `ma_liste` déclarée ci-dessus contient la valeur `NULL`, autrement dit elle ne pointe sur aucun élément. C'est de cette façon que les listes vides seront représentées.

Remarque : rien ne distingue un pointeur sur un élément d'une liste chaînée, c'est l'usage qui sera fait de ce pointeur qui en fera une liste chaînée (de même qu'un `char` est un entier et que l'usage qui en est fait permet de le considérer soit comme un entier, soit comme un caractère).

Exemple 1, liste chaînée contenant des entiers :

```
1 typedef struct _elt1 *Pelt1;
2 typedef struct _elt1{
3   int donnee;
4   Pelt1 suivant;
5 } Elt1;
6
7 Pelt1 ma_liste1=NULL;
```

Exemple 2, liste chaînée contenant des chaînes de caractères :

```
1 typedef struct _elt2 *Pelt2;
2 typedef struct _elt2{
3   char mot[30];
4   Pelt2 suivant;
5 } Elt2;
6
7 Pelt2 ma_liste2=NULL;
```

Exemple 3, liste chaînée contenant un entier, un nombre réel et un mot :

```
1 typedef struct _elt3 *Pelt3;
2 typedef struct _elt3{
3   int n;
4   float x;
5   char m[30];
6   Pelt3 suivant;
7 } Elt3;
8
9 Pelt3 ma_liste3=NULL:
```

7.3 FONCTIONS DE MANIPULATION DES LISTES CHAÎNÉES

La manipulation des listes chaînées est un peu plus complexe que celle des tableaux. Pour cette raison, les opérations les plus courantes sont habituellement regroupées dans des fonctions. Dans cette section seront présentées les fonctions implantant les opérations les plus courantes.

Dans la suite de cette section, la structure utilisée pour un élément sera la suivante :

```
1 typedef struct _un_element *P_un_element;
2
3 typedef struct _un_element
4         {
5         char mot[30];
6         P_un_element suivant;
7         } Un_element;
```

Il ne s'agit bien sûr que d'un cas particulier de liste chaînée. Les fonctions données dans la suite de cette section devront donc être adaptées aux besoins en changeant le traitement réalisé sur le ou les champs de données.

Ces fonctions sont assez courtes et ne présentent pas de difficultés particulières du point de vue de la syntaxe à partir du moment où les notions

de structure et de pointeurs sont bien maîtrisées. Cependant, il est parfois difficile de bien comprendre ce qu'elles font. Il est recommandé de bien les étudier et de vérifier leur comportement sur des exemples simples et sur les différents cas particuliers (selon les cas, liste vide, liste ne contenant qu'un seul élément, etc).

7.3.1 Création d'un élément

Cette fonction alloue la mémoire associée à un élément, initialise les différents champs et renvoie un pointeur sur le nouvel élément alloué par la fonction.

```
1  P_un_element creer_element(char *mot)
2          {
3          P_un_element el;
4
5          el = (P_un_element) malloc(sizeof(Un_element));
6          if (el == NULL) return NULL;
7          strcpy(el->mot,mot);
8          el->suivant = NULL;
9          return el;
10         }
```

7.3.2 Insérer en début de liste

Cette fonction ajoute un élément au début d'une liste. Il n'est pas nécessaire de traiter à part le cas d'une liste vide.

```
1  P_un_element inserer_element_debut(P_un_element pliste,
2                                     P_un_element el)
3          {
4           el->suivant = pliste;
5           return el;
6          }
```

7.3.3 Insertion d'un élément en fin de liste

Cette fonction ajoute un élément à la fin d'une liste. Pour cela, elle parcourt la liste jusqu'à atteindre le dernier élément, puis lui ajoute l'argument comme élément suivant. Il est à remarquer que pour cette fonction, il faut accéder aux champs de la liste afin de la parcourir. Il est donc nécessaire de traiter à part le cas de la liste vide.

```
1  P_un_element inserer_element_fin(P_un_element pliste,
```

```
2                                   P_un_element el)
3         {
4         P_un_element pl = pliste;
5
6         if (pliste == NULL)
7                 return el;
8
9         while (pl->suivant)
10                        {
11                        pl = pl->suivant;
12                        }
13        pl->suivant = el;
14
15        return pliste;
16        }
```

7.3.4 Insertion en place dans une liste triée

Cette fonction ajoute un élément dans une liste triée. Il faut donc parcourir la liste jusqu'à trouver le bon emplacement, puis insérer le nouvel élément en modifiant le chaînage. Comme il faut modifier l'élément précédant la position à laquelle insérer le nouvel élément, il faut garder en mémoire cet élément, ce qui peut se faire simplement en comparant la donnée associée à l'élément suivant celui sur lequel on se trouve. Il faut traiter à part le cas de la liste vide, mais aussi le cas de l'insertion en début de liste, puisque dans ce cas le pointeur sur le début de la liste est modifié.

La fonction suivante fait l'hypothèse que la donnée est une chaîne de caractères.

```
1 P_un_element inserer_en_place(P_un_element pliste,
2                   P_un_element el)
3         {
4         P_un_element  pl = pliste;
5
6         if (pliste == NULL)
7            return el;
8
9         if (strcmp(pl->mot, el->mot) >0)
10            {
11            el->suivant=pl;
12            return el;
13            }
14
15        while (pl->suivant)
```

```
16              {
17              if (strcmp(pl->suivant->mot, el->mot) > 0)
18                  break;
19                  pl = pl->suivant;
20              }
21          el->suivant = pl->suivant;
22          pl->suivant = el;
23          return pliste;}
```

7.3.5 Recherche d'un élément dans une liste

Cette fonction parcourt la liste chaînée à la recherche d'une donnée particulière et renvoie le pointeur sur l'élément qui contient cette donnée (s'il y en a plusieurs, l'adresse renvoyée est celle du premier élément rencontré). Si la donnée n'est pas contenue dans la liste, la fonction renvoie l'adresse `NULL`.

```
1  P_un_element rechercher_element(P_un_element pliste,
2                              char *mot)
3          {
4          while(pliste)
5                  {
6                  if (strcmp(pliste->mot, mot) == 0)
7                          return pliste;
8                  pliste = pliste->suivant;
9                  }
10         return NULL;
11         }
```

IMPORTANT : Cette fonction modifie l'argument `pliste`, qui contient l'adresse du début de la liste. Cette variable contient *une copie* de cette adresse, la modifier est donc sans conséquence sur la liste initiale.

ATTENTION, si `*pliste` avait été modifié (ou un champ de `pliste` en utilisant l'écriture '->'), cela aurait modifié la liste chaînée.

7.3.6 Suppression d'un élément dans une liste

La suppression d'un élément dans une liste nécessite de parcourir la liste pour trouver l'adresse de l'élément précédant celui que l'on cherche à supprimer. La fonction modifie ensuite le chaînage et renvoie le pointeur vers le premier élément de la liste. Il faut traiter à part le cas de la suppression du premier élément de la liste.

Remarque : cette fonction se contente de supprimer l'élément de la liste chaînée, elle ne libère pas la mémoire associée. La libération de mémoire devra être faite ailleurs.

```
1  P_un_element extraire_element(P_un_element pliste,
2          P_un_element pel)
3     {
4     P_un_element  pl = pliste;
5
6     if( pl == pel)
7          return pliste->suivant;
8
9     while (pl->suivant)
10         {
11         if (pl->suivant == pel)
12                 {
13                 pl->suivant = pel->suivant;
14                 break;
15                 }
16         pl = pl->suivant;
17         }
18    return pliste;
19 }
```

7.3.7 Libération d'une liste complète

Cette fonction détruit tous les éléments d'une liste chaînée, un par un. Il est à remarquer qu'un élément seul peut être libéré de cette façon à partir du moment où le champ `suivant` contient la valeur `NULL` (il s'agit dans ce cas d'une liste chaînée contenant un seul élément).

```
1  P_un_element detruire_liste(P_un_element pliste)
2          {
3          P_un_element  pl = pliste,pel;
4
5          while(pl)
6                  {
7                  pel = pl;
8                  pl = pl->suivant;
9                  free(pel);
10                 }
11         return NULL;
12         }
```

ATTENTION : dans ce cas, un élément complet est détruit avec un seul `free`. Dans le cas de structures plus complexes, certains champs peuvent nécessiter une libération explicite de mémoire. D'une manière générale, tous les champs ayant nécessité l'allocation explicite de mémoire (avec un `malloc`) doivent être libérés avec un `free` et uniquement ces champs-là.

7.3.8 Affichage du contenu d'une liste

Cette fonction affiche tous les mots d'une liste chaînée, un par ligne.

```
1 void afficher_liste(P_un_element pliste)
2         {
3         while(pliste)
4                 {
5                   printf("%s\n",pliste->mot);
6                   pliste=pliste->suivant;
7                 }
8         }
```

7.4 LISTES DOUBLEMENT CHAÎNÉES

L'insertion et la suppression d'un élément nécessite d'avoir accès à l'élément immédiatement précédent dans la liste chaînée de façon à modifier le chaînage. Si l'on dispose déjà, pour une raison ou pour une autre, du pointeur sur l'élément que l'on souhaite supprimer ou du pointeur sur l'élément avant lequel insérer un nouvel élément, il faut tout de même parcourir toute la liste pour retrouver l'adresse de l'élément précédent, ce qui peut être coûteux. Pour éviter cela, il est possible de définir des listes doublement chaînées dont chaque élément contiendra toujours l'adresse de l'élément suivant, mais également l'adresse de l'élément précédent. Tout élément de la liste peut alors être retrouvée à partir d'un élément quelconque de la liste. La déclaration d'un élément de liste doublement chaînée se fait de la façon suivante :

```
1 typedef struct _un_element *P_un_element;
2 typedef struct _un_element{
3
4    /* champ(s) de donnees */
5
6    P_un_element suivant;
7    P_un_element precedent;
8 } Un_element;
```

Pour l'exemple de liste avec des mots, cela donne ça :

```
typedef struct _un_element *P_un_element;

typedef struct _un_element
{
  char mot[30];
  P_un_element precedent;
  P_un_element suivant;
} Un_element;
```

Les listes doublement chaînées simplifient quelques fonctions, notamment la suppression d'un élément de la liste en évitant d'avoir à parcourir la liste. Les fonctions qui modifient le chaînage des éléments doivent également mettre à jour l'adresse de l'élément précédent. Les fonctions d'ajout en début et en fin de liste sont données à titre indicatif.

7.4.1 Supprimer un élément dans une liste

Pour supprimer un élément de la liste, il n'est plus nécessaire de la parcourir, il suffit de récupérer le pointeur sur l'élément précédent à partir du champ `precedent` et de le rediriger sur l'élément suivant.

```
P_un_element extraire_element(P_un_element pliste,
             P_un_element pel)
{

   if (pel->precedent!=NULL)
      pel->precedent->suivant = pel->suivant;

   if (pel->suivant!=NULL)
      pel->suivant->precedent = pel->precedent;

   /* cas particulier:
       on supprime le debut de la liste */
   if (pel->precedent==NULL)
      return pel->suivant;
   else
      return pliste;
}
```

7.4.2 Insérer en début de liste

L'insertion en début de liste est identique au cas de la liste simplement chaînée, au détail près qu'il faut initialiser les champs `precedent`.

```
1 P_un_element inserer_element_debut(P_un_element pliste,
2                     P_un_element el)
3 {
4    el->suivant = pliste;
5    el->precedent = NULL;
6
7    pliste->precedent=el;
8    return el;
9 }
```

7.4.3 Insérer un élément en fin de liste

L'insertion en fin de liste est là encore similaire, il suffit d'ajouter l'initialisation du champ `precedent` de l'élément qui vient d'être ajouté.

```
1  P_un_element inserer_element_fin(P_un_element pliste,
2                       P_un_element el)
3  {
4     P_un_element pl = pliste;
5
6     if (pliste == NULL)
7             return el;
8
9     while (pl->suivant)
10          {
11          pl = pl->suivant;
12          }
13    pl->suivant = el;
14    el->precedent=pl;
15    return pliste;
16 }
```

7.5 PILES ET FILES

Les piles et les files sont des structures de données très utilisées en informatique. Elles contiennent des données et leurs sont associées deux fonctions : une pour empiler un élément (l'élément est ajouté à la pile/file), cette fonction est appelée push, une pour dépiler un élément (l'élément est récupéré et enlevé de la pile/file), cette fonction est appelée pop.

On parle de file ou FIFO (First In First Out), lorsque l'élément dépilé est le plus ancien (le premier qui a été empilé), on parle aussi de façon équivalente de LILO (Last In Last Out).

On parle de pile ou LIFO (Last In First Out), lorsque l'élément dépilé est le plus récent (le dernier à avoir été empilé), on parle aussi de façon équivalente de FILO (First In Last Out).

7.5.1 Files FIFO

Les files FIFO peuvent être implantées sous la forme d'une liste chaînée. Ce qui compte est que l'empilement et le dépilement des données se fassent à des positions différentes. L'empilement peut être en tête de liste et le dépilement en fin de liste ou bien, de façon équivalente, l'empilement peut être en fin de liste et le dépilement en fin de liste.

```
1  typedef struct _File {
2    int data;
3    struct _File *suivant;
4  } File;
5
6  File *push_FIFO(File *file, int n) {
7    File *nelem=(File *)malloc(sizeof(File));
8    nelem->data=n;
9    nelem->suivant=file;
10   return nelem;
11 }
12
13 File *pop_FIFO(File *file,int *n) {
14   File *p=file, *pp=NULL;
15   if(p == NULL) {
16     printf("Erreur: appel de pop sur une file vide\n");
17     return NULL;
18   }
19
20   /* recherche du dernier element */
21   while (p->suivant!=NULL) {
22     pp=p;
23     p=p->suivant;
24   }
25   /* a la fin de la boucle,
26      p contient l'adresse du dernier element
27      et pp l'adresse de l'element precedent
28      (ou NULL) */
29
30   *n=p->data;
```

```
31
32    if (file->suivant==NULL) {
33      /* un seul element dans la liste */
34      free(pile);
35      return NULL;
36    }
37    else {
38      /* le test suivant n'est pas indispensable
39         si on arrive ici, la liste contient plus
40         d'un element aussi le dernier element a
41         necessairement un element precedent. */
42      if (pp) {
43        pp->suivant=NULL;
44      }
45      free(p);
46      return file;
47    }
48  }
49
50  int main() {
51    File *pFIFO=NULL;
52    unsigned int i;
53    int n;
54
55    for (i=0;i<10;i++) {
56      pFIFO=push_FIFO(pFIFO,i);
57    }
58
59    while(pFIFO) {
60      pFIFO=pop_FIFO(pFIFO,&n);
61      printf("pop_FIFO:␣%d\n",n);
62    }
63
64  }
```

L'exécution de ce programme donne l'affichage suivant :

```
pop_FIFO: 0
pop_FIFO: 1
pop_FIFO: 2
pop_FIFO: 3
pop_FIFO: 4
pop_FIFO: 5
pop_FIFO: 6
pop_FIFO: 7
```

```
pop_FIFO: 8
pop_FIFO: 9
```

7.5.2 Piles LIFO

Les piles LIFO peuvent être implantées sous la forme d'une liste chaînée. Ce qui compte dans ce cas est que la donnée empilée et la donnée dépilée soient à la même place. Comme l'ajout et la suppression en tête sont moins coûteux que l'ajout et la suppression en queue de liste (ces dernières nécessitant un parcours de la liste), ils sont préférés.

L'empilement d'un élément peut donc être implanté sous la forme d'une fonction d'ajout en tête. Le dépilement d'un élément peut donc être implanté sous la forme d'une fonction d'extraction en tête.

La structure de donnée à utiliser est la même que pour les piles FIFO.

```
1
2  Pile *push_LIFO(Pile *pile, int n) {
3    Pile *nelem=(Pile *)malloc(sizeof(Pile));
4    nelem->data=n;
5    nelem->suivant=pile;
6    return nelem;
7  }
8
9  Pile *pop_LIFO(Pile *pile,int *n) {
10   Pile *p;
11   if(pile == NULL) {
12     printf("Erreur:␣appel␣de␣pop␣sur␣une␣pile␣vide\n");
13     return NULL;
14   }
15   *n=pile->data;
16   p=pile->suivant;
17   free(pile);
18   return p;
19 }
20
21 int main() {
22   Pile *pLIFO=NULL;
23   unsigned int i;
24   int n;
25
26   for (i=0;i<10;i++) {
27     pLIFO=push_LIFO(pLIFO,i);
28   }
29
30   while(pLIFO) {
```

```
31      pLIFO=pop_LIFO(pLIFO,&n);
32      printf("pop_LIFO: %d\n",n);
33    }
34
35  }
```

L'exécution de ce programme donne l'affichage suivant :

```
pop_LIFO: 9
pop_LIFO: 8
pop_LIFO: 7
pop_LIFO: 6
pop_LIFO: 5
pop_LIFO: 4
pop_LIFO: 3
pop_LIFO: 2
pop_LIFO: 1
pop_LIFO: 0
```

8 ARBRES

Un arbre est une structure généralisant les listes chaînées. Au lieu de n'avoir qu'un seul successeur, dans un arbre, un élément peut disposer de plusieurs successeurs. Cette structure permet d'arranger différemment les données. Dans un certain nombre de cas de figure, il est plus pratique et plus efficace de ranger les données de cette façon. Plusieurs exemples concrets seront donnés. Ce chapitre commence par définir le vocabulaire associé aux arbres avant de présenter la façon de définir des arbres en C et de donner les fonctions les plus utiles pour cette structure. Le chapitre se termine sur plusieurs exemples courants d'utilisation d'arbres.

8.1 DÉFINITIONS

L'élément de base d'un arbre s'appelle le **noeud**. C'est une structure composée de :

— un (ou plusieurs) champ de donnée ;
— plusieurs pointeurs vers d'autres noeuds.

On utilise classiquement la terminologie suivante pour décrire les arbres :

— **Père**. Le père A d'un noeud B est l'unique noeud tel que :
  — un des sous-arbres contient B ;
  — hauteur(A) - hauteur(B) = 1.
— **Racine** La racine de l'arbre est l'unique noeud qui n'a pas de père.
— **fils** Les fils d'un noeud A sont les noeuds qui ont A pour père.
— **frères** Les frères d'un noeud A sont les noeuds qui possèdent le même père que A.
— **Sous-arbre** Un sous-arbre d'un noeud A est un arbre dont la racine est un fils de A.
— **Feuille** Une feuille est un noeud qui n'a pas de fils.
— **Branche** Une Branche est un arc qui relie deux noeuds.
— **Hauteur** Nombre maximum de branches entre la racine et une feuille.

Un arbre dont les noeuds ont au plus n fils est un arbre n-aire. Lorsque n vaut 2, l'arbre est dit binaire. Dans le cas particulier de l'arbre binaire, on utilise les termes de fils gauche et fils droit d'un noeud, ainsi que les notions de sous-arbre gauche et de sous-arbre droit.

8.2 IMPLANTATION EN C

Les arbres présentés dans la suite de ce chapitre sont des arbres binaires. Le cas des arbres généraux (dont les noeuds ont un nombre quelconque de fils) est évoqué à la fin du chapitre.

Pour définir une structure d'arbre binaire, comme pour les listes chaînées, il suffit de définir la structure permettant de stocker un noeud de l'arbre. Chaque noeud dispose de deux successeurs, il faut et il suffit donc de définir deux champs de type pointeur sur un noeud.

Exemple, arbre contenant des entiers :

```
1   typedef struct _un_noeud *P_un_noeud;
2
3   typedef struct _un_noeud
4        {
5        int data;
6
7        P_un_noeud fils_gauche;
8        P_un_noeud fils_droit;
9
10       } Un_noeud;
11
12  P_un_noeud mon_arbre;
```

D'une manière générale, une structure de noeud sera déclarée de la façon suivante :

```
1   typedef struct _un_noeud *P_un_noeud;
2
3   typedef struct _un_noeud
4        {
5        /* champs de donnees */
6
7        P_un_noeud fils_gauche;
8        P_un_noeud fils_droit;
9
10       /*optionnellement*/
11       P_un_noeud pere;
12       } Un_noeud;
13
14  P_un_noeud mon_arbre;
```

Remarque : par défaut, la navigation dans un arbre permet d'aller de la racine aux feuilles, mais comme dans le cas des listes doublement chaînées, il est possible de définir un champ `pere` qui permet également de naviguer des feuilles à la racine. Dans la suite, ce champ ne sera pas utilisé.

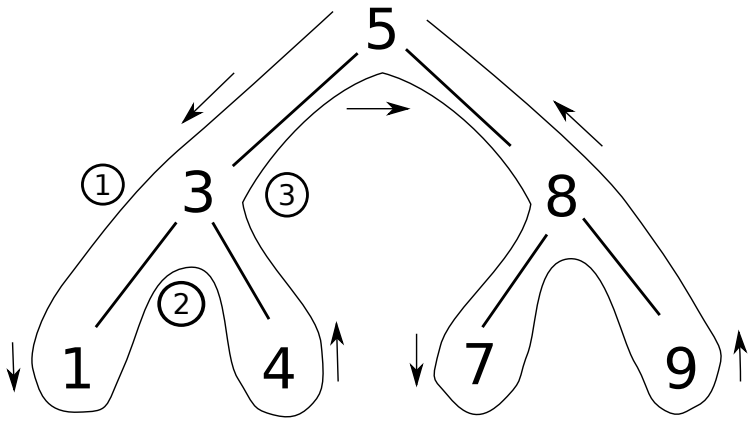


FIGURE 2 – Parcours en profondeur d'un arbre.

Dans la suite, la structure d'arbre considérée sera celle de l'exemple donné ci-dessus avec un entier comme donnée.

8.3 PARCOURS D'UN ARBRE

Le parcours d'un arbre est plus délicat que celui d'un tableau ou d'une liste. Pour ces structures, il y a en effet un seul successeur à un élément, aussi l'ordre de parcours est évident. Dans le cas d'un arbre binaire, il y a deux successeurs possibles, aussi faut-il choisir dans quel ordre les considérer et il faut prendre garde à n'oublier aucun élément.

Il y a plusieurs types de parcours différents. Tout d'abord, un parcours peut être en profondeur ou en largeur. Le parcours en profondeur va de la racine à une feuille puis remonte dans l'arbre pour considérer ensuite les autres feuilles. Le parcours en largeur considère d'abord tous les éléments qui sont à une hauteur donnée avant de passer à tous les noeuds qui sont à une hauteur supérieure de 1, etc (en commençant par la racine).

Il existe trois différents types de parcours en profondeur selon le moment où un élément est considéré pendant le parcours de l'arbre. En effet, si l'on suit l'arbre avec un tracé simulant le parcours, un noeud est rencontré à trois moments différents (voir figure 2 pour le noeud contenant la valeur 3). Si le noeud est considéré lors du premier passage, on parle de parcours préfixe (cas 1), s'il est considéré lors du second passage (cas 2), on parle de parcours infixe et s'il est considéré lors du troisième passage, on parle de parcours postfixe (cas 3).

Le parcours en profondeur préfixe de l'arbre représenté sur la figure 2 donne la séquence suivante : 5 3 1 4 8 7 9.

Le parcours en profondeur infixe de l'arbre représenté sur la figure 2 donne la séquence suivante : 1 3 4 5 7 8 9.

Le parcours en profondeur postfixe de l'arbre représenté sur la figure 2 donne la séquence suivante : 1 4 3 7 9 8 5.

Remarquez que le parcours indiqué commence par le sous-arbre gauche, on parle de parcours "main gauche". C'est le parcours le plus courant, cependant, il est bien sûr possible de commencer par le sous-arbre droit et de redéfinir ainsi les parcours préfixe, infixe et postfixe (parcours main droite).

La fonction de parcours peut appliquer n'importe quel traitement sur les noeuds. En guise d'exemple, les fonctions ci-dessous se contenteront d'afficher le contenu de l'arbre.

Ces différents types de parcours (version main gauche) sont présentés dans la suite.

8.3.1 Parcours en profondeur préfixe

Le parcours en profondeur préfixe traite le noeud courant, puis fait un appel récursif à la fonction de parcours sur le sous-arbre gauche puis sur le sous-arbre droit.

```
1      void disp(P_un_noeud arbre)
2      {
3      if (arbre)
4         {
5         printf("%d ", arbre->data);
6         disp(arbre->fils_gauche);
7         disp(arbre->fils_droit);
8         }
9      }
```

Le parcours d'un arbre se définit très simplement par une fonction récursive. Il peut également être écrit sous forme itérative, mais il faut dans ce cas utiliser une structure de pile LIFO pour mémoriser les noeuds restant à parcourir. Nous utiliserons dans la suite des fonctions similaires à celles vues précédemment, mais adaptées à la structure `P_un_noeud` (au lieu d'un `int`) :

```
1  void disp(P_un_noeud arbre)
2  {
3     P_un_noeud p_noeud = arbre;
4     Pile *pile = NULL;
5
6     if (p_noeud)
7     {
8        pile=push_LIFO(pile, p_noeud);
9        do
10       {
11         pile=pop_LIFO(pile, &p_noeud);
```

```
12          printf("%d␣", p_noeud->data);
13          if (p_noeud->droit) {
14               pile=push_LIFO(pile, p_noeud->droit);
15          }
16          if (p_noeud->gauche) {
17               pile=push_LIFO(pile, p_noeud->gauche);
18          }
19       }
20       while(pile);
21    }
22 }
```

Pour l'exemple, on a gardé le principe des fonctions push et pop tel que défini dans le chapitre 7. Cependant, pour cettre fonction, il faut redéfinir une structure et les fonctions push et pop pour une pile contenant non pas un entier mais un `P_un_noeud`.

8.3.2 Parcours en profondeur infixe

Le parcours en profondeur infixe fait un appel récursif à la fonction de parcours sur le sous-arbre gauche, puis traite le noeud courant et enfin fait un appel récursif sur le sous-arbre droit.

```
1    void disp(P_un_noeud arbre)
2       {
3       if (arbre)
4          {
5          disp(arbre->fils_gauche);
6          printf("%d␣", arbre->data);
7          disp(arbre->fils_droit);
8          }
9       }
```

8.3.3 Parcours en profondeur postfixe

Le parcours en profondeur postfixe fait un appel récursif à la fonction de parcours sur le sous-arbre gauche, sur le sous-arbre droit puis sur le noeud courant.

```
1    void disp(P_un_noeud arbre)
2       {
3       if (arbre)
4          {
5          disp(arbre->fils_gauche);
6          disp(arbre->fils_droit);
```

```
7           printf("%d␣", arbre->data);
8           }
9       }
```

Remarque : pour passer d'un type de parcours en profondeur à un autre, il suffit de changer la place de l'appel au `printf`.

8.3.4 Parcours en largeur

Le parcours en largeur se définit de façon itérative avec une file de type FIFO. Il suffit d'adapter la structure de file vue précédemment pour qu'elle contienne un `P_un_noeud` :

```
1  void disp(P_un_noeud arbre)
2  {
3     P_un_noeud p_noeud = arbre;
4     File *fifo = NULL;
5
6     if (p_noeud)
7     {
8        fifo=push_FIFO(fifo, p_noeud);
9        do
10       {
11          fifo=pop_FIFO(fifo, &p_noeud);
12          printf("%d␣", p_noeud->data);
13          if (p_noeud->gauche) {
14             fifo=push_FIFO(fifo, p_noeud->gauche);
15             }
16          if( p_noeud->droit){
17             fifo=push_FIFO(fifo, p_noeud->droit);
18        }
19       }
20       while(fifo);
21    }
22 }
```

La structure de données et le prototype des fonctions de push et de pop doivent être adaptés à une donnée de type `P_un_noeud`.

8.4 FONCTIONS DE MANIPULATION D'ARBRES

8.4.1 Création d'un noeud

La fonction de création d'un noeud alloue un noeud et initialise ses différents champs. Il est important de bien initialiser les champs de type pointeur à la valeur `NULL` de façon à limiter les risques d'erreur.

```
1  P_un_noeud creer_noeud(int data) {
2     P_un_noeud pa = malloc (sizeof(Arbre));
3     if (pa == NULL) {
4        fprintf(stderr,"Pb␣d'allocation.\n");
5        return NULL;
6     }
7     pa->data=data;
8     pa->gauche=NULL;
9     pa->droite=NULL;
10    return pa;
11 }
```

8.4.2 Ajout à la racine

La fonction d'ajout à la racine prend deux arbres existants et les insère comme sous-arbre d'un noeud nouvellement créé. Cette fonction permet de créer un arbre en partant des feuilles.

```
1  P_un_noeud ajouter_racine(int data,
2      P_un_noeud abg, P_un_noeud abd) {
3     P_un_noeud pa = creer_noeud(data);
4     pa->gauche=abg;
5     pa->droite=abd;
6     return pa;
7  }
```

8.4.3 Détruire un arbre

La fonction de destruction d'un arbre parcourt l'arbre complet et détruit chaque noeud.

ATTENTION : il ne faut pas oublier de détruire toute la mémoire qui a pu être allouée pour un noeud. Dans notre exemple, il n'y avait qu'un seul champ de type `int`, il n'y a donc pas de donnée supplémentaire à libérer un seul `free` suffit à libérer la mémoire allouée à un noeud.

```
1 void detruire_arbre(P_un_noeud pa) {
2    if (pa) {
3       detruire_arbre(pa->gauche);
4       detruire_arbre(pa->droite);
5       free(pa);
6    }
```

8.4.4 Afficher un arbre

Cette fonction affiche le contenu de l'arbre avec un parcours en profondeur de type préfixe :

```
1 void afficher_arbre(P_un_noeud pa) {
2    if (pa) {
3       printf("(%d␣",pa->data);
4       afficher_arbre(pa->gauche);
5       afficher_arbre(pa->droite);
6       printf(")");
7    }
8 }
```

8.4.5 Exemple

L'exemple ci-dessous crée un arbre contenant trois noeuds, les affiche et libère tous les noeuds alloués.

```
1 int main() {
2    P_un_noeud pa1=ajouter_racine(3,NULL,NULL);
3    P_un_noeud pa2=ajouter_racine(4,NULL,NULL);
4    P_un_noeud pa3=ajouter_racine(5,pa1,pa2);
5    afficher_arbre(pa3);
6    printf("\n");
7    detruire_arbre(pa3);
8 }
```

Après compilation et exécution, le programme affiche :

```
(5 (3 )(4 ))
```

Dans la suite de ce chapitre sont évoqués rapidement deux utilisations très classiques des arbres ainsi que les arbres généraux.

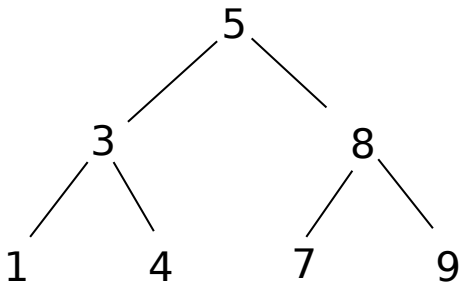
8.5 ARBRES BINAIRES DE RECHERCHE

Un arbre binaire de recherche est un arbre permettant d'ordonner des valeurs qui peuvent être comparées (que ce soit des nombres réels, des entiers ou une structure définie par l'utilisateur).

Tous les noeuds du sous-arbre de gauche ont une valeur inférieure à la valeur stockée sur le noeud courant.

Tous les noeuds du sous-arbre de droite ont une valeur supérieure ou égale à la valeur stockée sur le noeud courant.

L'arbre représenté sur la figure 2 était un exemple d'arbre binaire de recherche. Il est reproduit ici :



La structure définie précédemment permet de définir un tel arbre : il suffit en effet d'utiliser un entier comme donnée associée à un noeud. Pour en faire un arbre binaire de recherche, il faut ensuite définir les fonctions associées, à savoir une fonction d'ajout ainsi qu'une fonction de recherche dans l'arbre qui respecteront les règles énoncées ci-dessus.

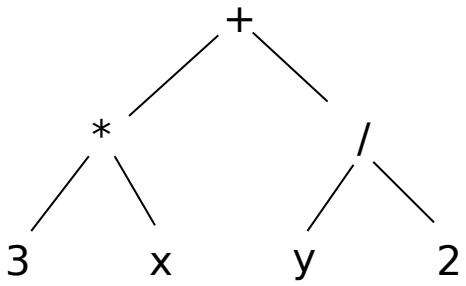
La fonction d'ajout se définit récursivement. Il faut comparer la valeur à ajouter au noeud courant. Si elle est inférieure, alors le nouveau sous-arbre gauche est le résultat de l'appel récursif à la fonction d'ajout en lui transmettant le fils gauche, si elle est supérieure, c'est pareil avec le sous-arbre droit et le fils droit.

La fonction de recherche se définit également récursivement de façon similaire. Si la valeur à chercher est inférieure au noeud courant, la recherche doit se poursuivre dans le sous-arbre gauche, sinon, c'est dans le sous-arbre droit.

Remarque : pour un arbre binaire de recherche, un parcours en profondeur main gauche infixe affiche les valeurs de l'arbre dans l'ordre croissant.

8.6 ARBRE D'EXPRESSION

Un arbre d'expression sert à représenter une expression arithmétique. Les feuilles contiennent des valeurs ou des variables et les noeuds contiennent des opérations. Exemple :



Cet arbre représente l'expression (3*x) + (y/2).

Cette structure est un peu plus difficile à définir que l'arbre binaire de recherche. Elle nécessite une structure de noeud différente de ce qui a été vu précédemment. Il faut en effet plusieurs champs pour stocker une constante, un nom de variable ou une opération. Il faut également un champ pour identifier le type du noeud (constante, variable ou opération).

Remarque : comme un seul type sera utilisé à la fois (constante ou variable ou opération), il est possible de définir une `union`, dont le fonctionnement est similaire à un `struct`, au détail prêt que les champs sont superposés au lieu d'être disposés en mémoire les uns à la suite des autres. Cela permet donc d'économiser de la mémoire (la taille d'une variable de type `union` est le maximum des tailles de chacun de ses champs, au lieu de la somme pour un `struct`) . Dans ce cas aussi, cependant, le champ permettant d'identifier le noeud est nécessaire.

Il faut ensuite définir les fonctions qui vont permettre de manipuler cet arbre, que ce soit pour le construire ou pour évaluer une expression.

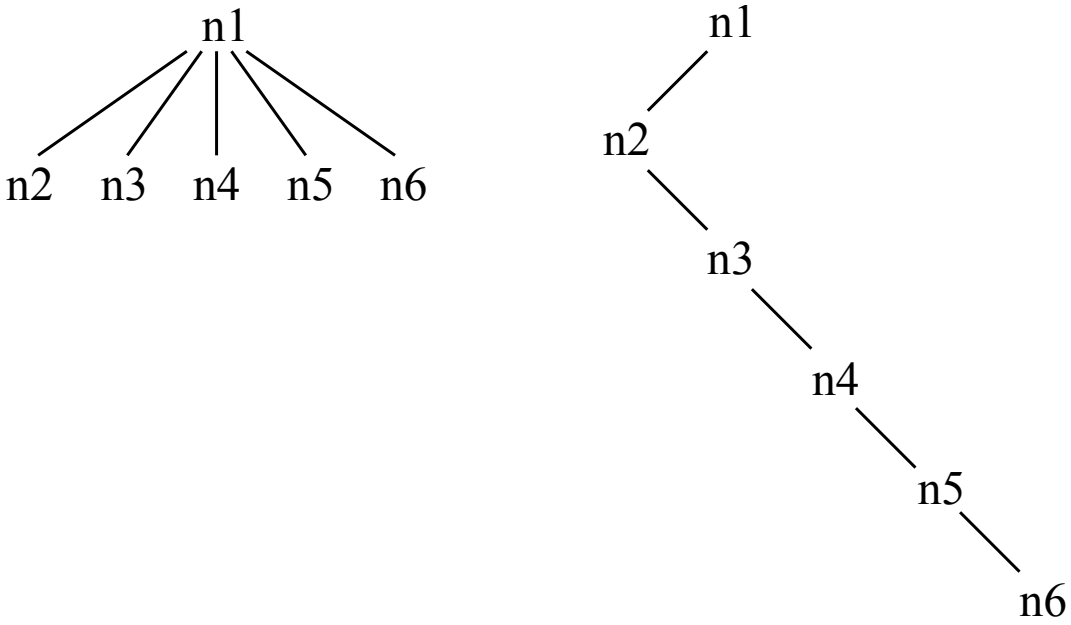
La construction de l'arbre est relativement aisée si elle se fait à partir d'une expression donnée en écriture préfixée (opérateur opérande1 opérande2). Dans ce cas, il faut lire le contenu du prochain noeud, le créer et l'initialiser en testant de quel type il est (nombre, opérateur ou variable, ce qui peut se faire avec des `if`). Selon le type, la lecture de la branche sera finie (si le type est constante ou variable) ou au contraire nécessitera la lecture de deux nouveaux opérandes. Si l'expression est en écriture postfixée, il faut utiliser une pile, empiler les opérandes lus et dépiler lorsqu'un opérateur est lu.

L'évaluation nécessite un parcours de l'arbre avec un traitement dépendant du type de noeud : l'évaluation d'une variable ou d'une constante renvoie la valeur associée, l'évaluation d'une opération nécessite l'évaluation de ces deux opérandes avant de calculer l'opération et de renvoyer la valeur.

8.7 ARBRES GÉNÉRAUX

Les arbres généraux sont des arbres dont les noeuds ont un nombre arbitraire de fils. Ils peuvent être implantés en utilisant des listes chaînées pour stocker les fils.

Il est aussi possible de les implanter avec des arbres binaires. Pour cela, on peut utiliser la convention suivante : le fils gauche d'un noeud est un des fils de l'arbre général, le fils droit est un frère. Exemple (Gauche : arbre général. Droite : représentation de l'arbre général avec un arbre binaire) :



Un arbre général peut donc utiliser la structure d'arbre binaire telle qu'elle a été définie précédemment. Comme précédemment, il faut associer à cette structure des fonctions qui lui donneront un comportement similaire à celui d'un arbre général.