C AVANCÉ

LAGNIEZ JEAN-MARIE (LAGNIEZ@CRIL.FR)

Table des matières

Chapitre 1

Introduction à la programmation en C

1.1 Historique

Le C a été développé en 1972 par Dennis Richie et Kenneth Thompson, chercheurs aux Bell Labs, dans l'optique de développer un système d'exploitation UNIX sur un DEC PDP-11. Kenneth Thompson avait développé le prédécesseur direct de C, le langage B, qui est lui-même largement inspiré de BCPL. Dennis Ritchie a fait évoluer le langage B dans une nouvelle version suffisamment différente, en ajoutant notamment les types, pour qu'elle soit appelée C1.
En 1978, Brian Kernighan et Dennis Richie publient la définition classique du C dans le livre *The C Programming language* (Le langage C). Le C devenant de plus en plus populaire dans les années 80, plusieurs groupes mirent sur le marché des compilateurs comportant des extensions particulières. En 1983, l'ANSI (American National Standards Institute) décida de normaliser le langage ; ce travail s'acheva en 1989 par la définition de la norme ANSI C. Celle-ci fut reprise telle quelle par l'ISO (International Standards Organization) en 1990. C'est ce standard, ANSI C, qui nous allons étudier dans ce cours.

1.2 La compilation d'un programme C

Le C est un langage compilé, cela signifie qu'un programme C est décrit par un fichier texte (fichier source). Voici un exemple de programme C (`hello.c`) :

```c
#include <stdio.h>

int main(int argc, char ** argv)
{
  printf("Hello world!\n");
  return 0;
}
```

Ce fichier ne peut pas directement être exécuté par le microprocesseur, il faut le traduire en langage machine (assembleur). Cette opération est effectuée par un programme appelé compilateur, nous utiliserons le compilateur `gcc`. La compilation d'un programme C se décompose en 4 phases successives :

1. **Le traitement par le préprocesseur** : le fichier source est analysé par le préprocesseur qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers source ...). Il est possible d'afficher cette étape en utilisant l'option `-E` du compilateur `gcc`.

   ```
   $ gcc -E hello.c
   ...
   ```

2. **La compilation** : elle traduit le fichier généré par le pré-processeur en langage assembleur, c'est-à-dire en langage machine pouvant être interprété par un humain. `gcc` permet d'obtenir le code assembleur d'un programme C :

```
$ gcc -S hello.c
$ cat hello.s
...
```

3. **L'assemblage** : cette opération transforme le code assembleur en un fichier binaire, c'est-à-dire en instructions directement compréhensibles par le processeur (impossible à lire par un humain). Généralement, la compilation et l'assemblage se font dans la foulée (sauf si l'on spécifie explicitement que l'on veut le code assembleur). Le fichier ainsi généré est appelé fichier objet et il peut être obtenu à l'aide du compilateur `gcc` de la manière suivante :

   ```
   $ gcc -c hello.c
   ```

   Il est possible de récupérer le code assembleur directement sur le fichier objet via l'utilisation de la commande `objdump` :

   ```
   $ objdump -d hello.o
   ```

4. **L'édition de liens** : vos programmes seront généralement séparés en plusieurs fichiers source, pour des raisons de clarté mais aussi parce qu'il fait aussi appel à des librairies externes (les librairies standards, interface graphique, ...). Une fois chaque code source assemblé, il faut donc lier entre eux les différents fichiers objets. L'édition de liens produit alors un fichier exécutable. La commande `gcc` permettant d'obtenir l'exécutable est :

   ```
   $ gcc hello.c
   ```

   Dans ce cas le fichier exécutable `a.out` est généré :

   ```
   $ ls -l a.out
   -rwxr-xr-x 1 lagniez lagniez 16840 15 juin  15:07 a.out
   $ ./a.out
   Hello world!
   ```

   Il est aussi possible de générer le fichier exécutable directement à partir du code objet de la manière suivante :

   ```
   $ gcc -c hello.c
   $ gcc hello.o
   $ ./a.out
   ```

   `gcc` permet aussi de nommer directement l'exécutable à sa guise :

   ```
   $ gcc -o exec hello.c
   $ ./exec
   ```

`gcc` comporte beaucoup d'options que nous ne verrons pas lors de ce cours. Cependant il y en a une que nous allons rapidement utilisée, c'est l'option `-g`. Cette option sert à déboguer un programme, il est nécessaire de pouvoir obtenir certaines informations le concernant, comme le nom et le type des variables, ainsi que les numéros de lignes correspondant aux instructions. Lorsque votre programme est compilé avec cette option, il est possible d'utiliser un outils de débogage tel que `gdb`, le débogueur de `GNU`.

```
$ gcc -g -o exec hello.c
$ gdb ./exec
GNU gdb (Debian 12.1-3) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.
```

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./exec...
(gdb) run
Starting program: /home/lagniez/Works/Enseignement/c-av-c/cours/exemples/exec
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Hello world!
[Inferior 1 (process 10583) exited normally]
(gdb)
```

Nous n'allons pas présenter `gdb` en détail maintenant, nous y reviendrons tout au long de ce cours. Nous pouvons par exemple affiche le code assembleur de notre programme :

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x0000555555555139 <+0>:    push   %rbp
   0x000055555555513a <+1>:    mov    %rsp,%rbp
   0x000055555555513d <+4>:    sub    $0x10,%rsp
   0x0000555555555141 <+8>:    mov    %edi,-0x4(%rbp)
   0x0000555555555144 <+11>:   mov    %rsi,-0x10(%rbp)
   0x0000555555555148 <+15>:   lea    0xeb5(%rip),%rax        # 0x555555556004
   0x000055555555514f <+22>:   mov    %rax,%rdi
   0x0000555555555152 <+25>:   call   0x555555555030 <puts@plt>
   0x0000555555555157 <+30>:   mov    $0x0,%eax
   0x000055555555515c <+35>:   leave
   0x000055555555515d <+36>:   ret
End of assembler dump.
```

1.3 Les composants d'un programme C

Revenons un instant sur le programme `hello.c` et voyons se qu'il nous raconte. Tout programme C, quelle que soit son objectif, est constitué de fonctions et de variables. Les fonctions contiennent des instructions qui indique les opérations à effectuer, et les variables stockent les valeurs utilisées. Les fonctions sont en quelques sortes des sous-programmes que nous pouvons nommer. Généralement vous pouvez donner n'importe quel nom à vos fonctions [1], à part `main` qui est un cas particulier puisque c'est la fonction principale du programme qui sera exécuté lorsque vous lancerez ce dernier. Cela implique que vous ne pouvez n'avoir qu'une seule fonction `main` dans votre programme!
Il est possible d'inclure d'autres fichiers dans votre programme.

```
1 #include "nom_du_fichier"
```

Généralement ces fichier sont des *headers* qui contiennent la définition de fonctions et structures que vous souhaitez utiliser dans votre programme. Dans notre exemple, la première ligne inclut certaines informations sur la bibliothèque standard d'entrées-sorties. C'est le préprocesseur qui aura la tâche d'inclure les informations dans le fichier (le résultat de l'inclusion du fichier peut être observé lorsqu'on applique l'option `-E` à la commande `gcc`.
Pour le moment nous avons vu l'inclusion de fichiers avec des chevrons et pas des guillemets.

```
1 #include <nom_du_fichier>
```

En fait, la notation chevron est réservé pour les *headers* système et la notation guillemets est réservé pour les *headers* de notre programme (il est possible d'ajouter des chemins pour les *header* système, mais nous verrons cela plus en détail lorsque nous verrons la compilation modulaire (**??**)).
Pour « communiquer » avec une fonction, il est possible de lui transmettre une liste de paramètres appelés arguments et de récupérer (si nous le souhaitons) une seul valeur. Dans notre exemple la fonction `main` prend en paramètre deux arguments et retourne un entier. Nous appelons aussi la fonction `printf`

1. Ce n'est pas tout à fait vrai puisqu'il faut quand même respecter les conventions de nommage, nous verrons cela plus en détail plus tard.

avec comme paramètre une chaîne de caractère. Cependant, comme le montre la page `man` de la fonction `printf`, cette fonction retourne un élément que nous n'allons pas récupérer (ce qui est possible) et ses arguments sont un peu particulier ... nous verrons cela plus en détail lorsque nous verrons les définitions de fonctions (**??**).

Pour être plus précis, un programme C est constitué de composantes élémentaires :

— **Les commentaires :** Il débute par /* et se termine par */. Nous ne pouvons pas imbriquer des commentaires. Quand nous mettons en commentaire un morceau de programme, il faut faire attention que celui-ci ne contienne pas de commentaires! Re-voilà notre programme commenté :

```
#include <stdio.h>              /* inclut des informations sur la bibliothèque standard */

int main(int argc, char ** argv) /* définition de la fonction main */
{
  printf("Hello world!\n");      /* main appelle la fonction printf */
  return 0;                     /* retourne 0 au programme exécutant */
}
```

Nous pouvons aussi utiliser les commentaires à la C++ // lorsque nous souhaitons commenter une ligne. Par exemple :

```
/*
  Un super programme très poli !
  Il est aussi commenté, peut être un peu trop :D
*/
#include <stdio.h>              // inclut des informations sur la bibliothèque standard

int main(int argc, char ** argv) // définition de la fonction main
{
  printf("Hello world!\n");      // main appelle la fonction printf
  return 0;                     // retourne 0 au programme exécutant
}
```

L'ensemble des commentaires est supprimé par le préprocesseur (vous pouvez voir cela à l'aide de l'option de compilation `-E` de `gcc`).

— **Les identificateurs** : Il servent à donner un nom à une entité du programme, un identificateur peut désigner :
  — un nom de variable ou de fonction ;
  — un type défini par `typedef`, `struct`, `union` ou `enum` ;
  — une étiquette.

  Un identificateur est une suite de caractères parmi :
  — les lettres (minuscules ou majuscules, mais non accentuées) ;
  — les chiffres ;
  — le underscore (_).

  Le premier caractère d'un identificateur ne peut pas être un chiffre. Il est aussi déconseillé d'utiliser comme premier caractère d'un identificateur le caractère _ car il est souvent utilisé pour définir les variables globales de l'environnement C. Le langage C respect la case, c'est-à-dire que les minuscules et les majuscules sont différenciées. Le compilateur peut tronquer les identificateurs au-delà d'une certaine longueur (c'est au moins 31 caractères pour les compilateurs modernes [2]).

— **Les mots-clefs :**
  Un certain nombre de mots-clefs, sont réservés pour le langage lui-même et ne peuvent pas être utilisés comme identificateurs. L'ANSI C compte 32 mots clefs :
  Nous pouvons ranger les mots-clefs de la manière suivante :
  — en bleu les spécificateurs de stockage

2. Le standard dit que les identificateurs externes, c'est-à-dire ceux qui sont exportés à l'édition de lien, peuvent être tronqués à 6 caractères, mais tous les compilateurs modernes distinguent au moins 31 caractères

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

TABLE 1.1 – Mots-clefs en C

— en rouge les qualificateurs de type
— en vert les spécificateurs de type
— en jaune les instructions de contrôle
— en violet le reste

Si nous utilisons un mot-réservé pour nommer, la compilation sera stoppée avec une erreur qui nous indiquera pourquoi cela n'a pas fonctionné :

```
$ cat hello_test.c
#include <stdio.h>

int main(int argc, char ** argv)
{
  int break = 10;
  printf("Hello world!\n");
  return 0;
}

$ gcc hello_test.c
hello_test.c: In function 'main':
hello_test.c:7:7: error: expected identifier or '(' before 'break'^
        5 |   int break = 10;
```

— **Les constantes :** ce sont des valeurs qui apparaissent littéralement dans le code source d'un programme, le type de la constante étant déterminé par la façon dont la constante est écrite. Les constantes peuvent être de 4 types : entier, flottant (nombre réel), caractère ('a', 'b', ...), énumération (enum). Ces constantes vont être utilisées, par exemple, pour initialiser une variable (nous verrons cela plus en détail dans le chapitre suivant 2).
— **Les (constantes) chaînes de caractères :** c'est une séquence de caractères, éventuellement vide, placée entre guillemets, comme par exemple :

```
"Je suis une chaîne"
```

ou encore

```
""
```

Il faut noter que les guillemets ne font pas partie de la chaîne, ils ne servent qu'à la délimiter. Nous pouvons utiliser le caractère d'échappement \ lorsque nous souhaitons imprimer le caractère " dans une chaîne. Il est aussi possible de concaténer des constantes de type chaîne à la compilation :

```
"Je suis ""une chaîne"
```

équivaut à

```
"Je suis une chaîne"
```

Cela peut être utilise lorsque nous souhaitons découper une longue chaîne de caractères afin de l'écrire sur plusieurs lignes :

```
"Je suis aussi une chaîne"\
"... mais je suis un peu plus longue"\
"... vraiment plus longue."
```

Atention : le caractère d'échappement est obligatoire, car lorsque nous retournons à la ligne nous imprimons le caractère de retour à la ligne et non le caractère vide !
— **Les opérateurs :** il en existe de plusieurs types (que nous verrons en détails dans le chapitre **??**) :
— opérateur d'affectation (=)

- opérateurs arithmétiques (+, -, ...)
- opérateurs relationnels (>, <, ...)
- opérateurs logiques booléens (||, && et !)
- opérateurs logiques bit à bit (|, &, ...)
- opérateurs d'affectation composée (+=, -=, ...)
- operateurs d'incrémentation et de décrémentation (++ et –)
- opérateur virgule
- opérateur conditionnel ternaire (test ? op1 : op2)
- opérateur de conversion de type (cast)
- opérateur adresse (&)

— **Les signes de ponctuation :** Les signes de ponctuation en langage C ont une signification syntaxique et sémantique pour le compilateur mais ne spécifient pas d'eux-mêmes une opération qui produit une valeur. Certains signes de ponctuation, seuls ou combinés, peuvent également être des opérateurs C ou être significatifs pour le préprocesseur. Tous les caractères suivants sont considérés comme des signes de ponctuation :

```
! % ^ & * ( ) - + = { } | ~
[ ] \ ; : < > ? , . / # ' "
```

Les ponctuateurs [ ], ( ), et { } doivent apparaître en paires. Chaque instruction d'un programme C doit se terminer par le signe de ponctuation ";". Le signe de ponctuation ";" marque donc soit la fin d'une instruction, soit la fin d'une liste de déclarations.

Chapitre 2

Les types de bases

Contrairement à un langage comme Python, le C est un langage typé. Ainsi toute variable, constante ou fonction est d'un type précis. Particulièrement, le type d'un objet définit la façon dont il est représenté en mémoire. Quand une variable est déclarée, il lui est attribué une adresse. Cette variable correspondra à une zone mémoire dont la longueur (le nombre d'octets) est fixée par le type. Le nombre d'octets utilisé pour représenter un type en mémoire dépendra du compilateur. Néanmoins, la norme ANSI spécifie un certain nombre de contraintes sur les types de bases, que sont les caractères, les entiers et les flottants (nombres réels). Ils sont désignés par les mots-clefs suivants :

```
char
int
float double
short long unsigned
```

2.1 Le type caractère

En C il n'y a pas vraiment de notion de caractère à proprement parlé, en fait nous utilisons différent types d'entiers dépendant du nombre d'octets nécessaires pour les représenter ainsi que de leur format, c'est-à-dire si ils sont signés ou non. Ainsi, bien que le mot-clef `char` désigne une variable de type caractère, il n'est en général qu'un objet de type entier codé sur un octet. C'est ensuite à l'exécution que la machine utilisée pourra mettre en correspondance l'entier et le caractère que nous souhaitons imprimer. Le jeu de caractères de base utilisé correspond généralement au codage ASCII (sur 7 bits).

```
#include <stdio.h>

int main(int argc, char** argv) {
  char c = 0;
  while (c >= 0) {
    printf("%c %d\n", c, c);
    c = c + 1;
  }

  return 0;
}
```

Ici, nous avons fait la supposition que le `char` était codé sur un entier et donc que cela fonctionnerait, mais cela ne peut ne pas être le cas ! En fait, les valeurs maximales et minimales des différents types entiers sont définies dans la librairie standard `limits.h`. Le mot-clef `sizeof` permet de récupérer la taille d'un type ou d'un objet, le résultat étant un entier égal au nombre d'octets nécessaires pour stocker le type ou l'objet.

```
#include <stdio.h>

int main(int argc, char** argv) {
  char c = 0;
```

```
5   printf("j'ai un char qui est codé sur %d %d\n", sizeof(char), sizeof(c));
6   return 0;
7 }
```

Étant donné le caractère universel de l'informatique s'exportant, il devenait nécessaire de pouvoir employer d'autres caractères que ceux présents dans la table ASCII. C'est ainsi que d'autres tables ont commencé à faire leur apparition, quasiment toutes basées sur la table ASCII et l'étendant comme le latin-1 ou ISO 8859-1 pour les pays francophones, ISO 8859–5 pour l'alphabet cyrillique, ISO 8859–6 pour l'alphabet arabe ... le bazar!

Afin d'éviter la multiplication des tables, il a été décidé de construire une table unique, universelle qui contiendrait l'ensemble des alphabets : la table Unicode. Il existe trois encodages majeures pour l'Unicode : UTF-32, UTF-16 et UTF-8. Il faudra donc faire attention à l'encodage des fichiers que vous aurez à parser à l'avenir!

Le langage définit des constantes pour assigner des caractères dans le code source. Pour désigner un caractère imprimable, il suffit de le mettre entre apostrophes ('A' ou '€'). Les seuls caractères imprimables qu'on ne peut pas représenter de cette façon sont l'antislash et l'apostrophe, qui sont respectivement désignés par \\ et \'. Les caractères non imprimables peuvent être désignés par '\code-octal' où valeur est le code en octal du caractère. On peut aussi écrire '\xcode-hexa' où code-hexa est le code en hexadécimal du caractère (cf. page 15). Par exemple, '\33' et '\x1b' désignent le caractère escape. Les caract'eres non-imprimables les plus fréquents disposent aussi d'une notation plus simple :

`\n` nouvelle ligne	`\r` retour chariot
`\t` tabulation horizontale	`\f` saut de page
`\v` tabulation verticale	`\a` signal d'alerte
`\b` retour arrière	

2.2 Les types entiers

Un nombre entier est un nombre sans virgule, ils sont signés par défaut, cela signifie qu'ils comportent un signe. Pour stocker l'information concernant le signe (en binaire), les ordinateurs utilisent le complément à deux.

Le mot-clef désignant le type entier est `int`, un objet de type `int` est représenté par un mot « naturel » de la machine utilisée, c'est-à-dire 32 bits pour un PC Intel. Le type int peut être précédé d'un attribut de précision (short ou long) et/ou d'un attribut de représentation (`unsigned`).

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv) {
4   printf("%d %d %d %d %d\n", sizeof(char), sizeof(short), sizeof(int),
5                  sizeof(long), sizeof(long long));
6   return 0;
7 }
```

En complément à deux, le bit de poids fort d'un entier est son signe. Un entier positif est donc représenté en mémoire par la suite de 32 bits dont le bit de poids fort vaut 0 et les 31 autres bits correspondent à la décomposition de l'entier en base 2. Un entier négatif est, lui, représenté par une suite de 32 bits dont le bit de poids fort vaut 1 et les 31 autres bits correspondent. Un `int` peut donc représenter un entier entre -2^{31} et $(2^{31} - 1)$.

L'attribut `unsigned` spécifie que l'entier n'a pas de signe. Un `unsigned int` peut donc représenter un entier entre 0 et $(2^{32} - 1)$. Nous utiliserons donc un des types suivants en fonction de la taille des données à stocker :

`signed char`	$[-2^7, 2^7[$
`unsigned char`	$[0, 2^8[$
`short int`	$[-2^{15}, 2^{15}[$
`unsigned short int`	$[0, 2^{16}[$
`int`	$[-2^{31}, 2^{31}[$
`unsigned int`	$[0, 2^{32}[$
`long int`	$[-2^{63}, 2^{63}[$
`unsigned long int`	$[0, 2^{64}[$

Attention : Pour obtenir des programmes portables, nous ne présumerons jamais de la taille d'un objet de type entier. Nous utiliserons toujours une des constantes de `limits.h` ou le résultat obtenu en appliquant l'opérateur `sizeof`.

Une constante entière peut être représentée de 3 manières suivant la base dans laquelle elle est écrite :

- décimale : par exemple, 61 et 1664 sont des constantes entières décimales.
- octale : représente un entier en base 8. Les constantes octales commencent par un zéro, par exemple les entiers 0 et 255 sont respectivement 00 et 0377 en octale.
- hexadécimale : représente un entier en base 16. Les constantes hexadécimales commencent par 0x ou 0X. Par exemple, les entiers 14 et 255 sont respectivement 0xe et 0xff en hexadécimale.

Par défaut, une constante décimale est représentée avec le format interne le plus court. Il faut donc faire attention lorsque nous réalisons des opérations avec des constantes. Par exemple le code suivant n'a pas du tout le comportement souhaitait :

```c
#include <stdio.h>

int main(int argc, char** argv) {
  long v = 2000000000 + 2000000000;
  printf("%ld\n", v);
  return 0;
}
```

Nous pouvons cependant spécifier explicitement le format d'une constante entière en la suffixant par u ou U pour indiquer qu'elle est non signée, ou en la suffixant par l ou L pour indiquer qu'elle est de type long. Par exemple : 42 (int), 042 (int octal), 0x42 (int hexadécimal), 42L (long), 42U (unsigned int) et 42UL (unsigned long int).

```c
#include <stdio.h>

int main(int argc, char** argv) {
  long v = 2000000000L + 2000000000L;
  printf("%ld\n", v);
  return 0;
}
```

2.3 Les types flottants

Les types float, double et long double servent à représenter des nombres en virgule flottante. Ils correspondent aux différentes précisions possibles.

```c
#include <stdio.h>

int main(int argc, char** argv) {
  printf("%d\n", sizeof(float));
  printf("%d\n", sizeof(double));
  printf("%d\n", sizeof(long double));
  return 0;
}
```

Les flottants sont généralement stockés en mémoire sous la représentation conforme aux normes numériques IEEE. Les nombres réels sont des nombres à virgule flottante, c'est-à-dire un nombre dans lequel

la position de la virgule n'est pas fixe, et est repérée par une partie de ses bits (appelée l'exposant), le reste des bits permettent de coder le nombre sans virgule (la mantisse).
Les nombres de type `float` sont codés sur 32 bits dont :

23 bits pour la mantisse	8 bits pour l'exposant	1 bit pour le signe

Les nombres de type `double` sont codés sur 64 bits dont :

52 bits pour la mantisse	11 bits pour l'exposant	1 bit pour le signe

Les nombres de type `long double` sont codés sur 80 bits dont :

64 bits pour la mantisse	15 bits pour l'exposant	1 bit pour le signe

La précision des nombres réels est approchée. Elle dépend par le nombre de positions décimales, suivant le type de réel elle sera au moins :

— 6 chiffres pour le type `float`
— 15 chiffres pour le type `double`
— 17 chiffres pour le type `long double`

Les constantes réelles sont représentées par la notation classique par mantisse et exposant. L'exposant est introduit par la lettre e ou E. Par défaut, une constante réelle est représentée avec le format du type `double`. Nous pouvons cependant spécifier la représentation influer en ajoutant un des suffixes f (F) ou l (L). Le suffixe f force la représentation de la constante sous forme d'un `float`, le suffixe l force la représentation sous forme d'un long double. Par exemple : 12.34 est un `double`, 12.3e-4 un `double`, 12.34F un `float` et 12.34L un `long double`.

2.4 Utilisation de `gdb` pour afficher le contenu d'une variable

Avec tout ce que nous venons de voir, vous êtes capables de lancer et d'arrêter l'exécution d'un programme. Mais cela suffit rarement à trouver la cause d'un plantage. Maintenant, lorsque le programme plante, nous aimerions connaître l'état de certaines variables, pouvoir les modifier, ou encore savoir exactement à quel moment le programme a planté.
Il est possible d'utiliser l'utilitaire `gdb` pour afficher le contenue d'une variable. Pour cela il nous faut d'abord stopper l'exécution du programme à un point donné en ajoutant soit un *breakpoint* ou un *watchpoint*.
Lorsque nous posons un *breakpoint*, `gdb` s'arrête juste avant l'instruction qui suit ce dernier. Cela signifie par exemple que si vous demandez à `gdb` de s'arrêter à la ligne 17, il va s'arrêter juste au début de la ligne, sans l'exécuter. Pour poser un *breakpoint* il existe plusieurs manières :

— `break [position]` permet d'ajouter un *breakpoint* par rapport à une position. Si on ne donne pas de position, `gdb` s'arrêtera à la prochaine instruction qui sera exécutée. Sinon on peut spécifier l'endroit où le programme sera arrêté :
  — en donnant un numéro de ligne (`break 42`) du fichier courant
  — en donnant un décalage en lignes par rapport à la prochaine instruction (`break +5` s'arrêter 5 lignes après la prochaine instruction)
  — en donnant un nom de fichier et un numéro de ligne (`break hello.c:5`)
  — en donnant un nom de fonction (`break main`)
  — en donnant un nom de fichier et de fonction (`break hello.c:main`)
  — en donnant une adresse (`break *0x0000555555555191`, l'adresse peut être un nombre, une variable, une expression ...).
— `break [position] if condition` permet de préciser une condition d'arrêt. `gdb` va alors évaluer (calculer) cette condition juste avant de s'arrêter, pour savoir s'il doit ou non continuer (`break 42 if a == 42`).
— `clear [position]` permet de supprimer le *breakpoint* correspondant à la position indiquée. Comme précédemment, ne pas indiquer de position suppose qu'il s'agit de l'instruction suivante.

— `info breakpoints` affiche la liste des *breakpoints*.

Il est possible de demander à `gbd` de s'arrêter lorsqu'une variable est lue ou modifiée. Pour cela nous utilisons la commande `watch` de `gdb` de la manière suivante :

— `watch variable` indique à `gdb` de s'arrêter dès que la variable est modifiée. L'instruction qu'il vous affiche correspond donc à celle qui sera exécutée juste après (et non pas à celle qui a modifié la variable, puisqu'elle a déjà été exécutée). Vous pouvez également utiliser une expression, par exemple "`a*b`", pour indiquer à `gdb` de ne s'arrêter que lorsque (`a*b`) aura été modifié. Mais il faut bien noter que pour surveiller une expression, `gdb` est obligé de surveiller chaque variable de cette expression et d'évaluer le résultat.

— `rwatch variable` indique à `gdb` de s'arrêter dès que la variable est accédée par le programme.

— `awatch variable` indique à `gdb` de s'arrêter aussi bien lorsque la variable est modifiée que lorsqu'elle est lue.

— `info watchpoints` affiche la liste des *watchpoints*.

Vous êtes à présent capable de lancer et d'arrêter l'exécution d'un programme. Lorsque le programme plante, nous aimerions connaître l'état de certaines variables. Pour cela rien de plus simple, il suffit d'utiliser la commande `print` en précisant le nom de la variable. Attention, il faut que la variable existe au moment où vous demandez l'affichage. Un petit exemple :

```
$ cat prog.c
#include <stdio.h>

int main(int argc, char** argv) {
  int c;
  c = 1;
  c = 2;
  c = 3;
  return 0;
}
$ gcc -g -o exec prog.c
$ gdb ./exec
...
(gdb) break main
Breakpoint 1 at 0x1130: file prog.c, line 5.
(gdb) run
Starting program: /home/lagniez/Works/Enseignements/c-av-c/cours/exemples/exec

Breakpoint 1, main (argc=1, argv=0x7fffffffe038) at prog.c:5
5   c = 1;
(gdb) print c
$1 = 0
(gdb) next
6   c = 2;
(gdb) print c
$2 = 1
(gdb) next
7   c = 3;
```

Chaque fois que vous affichez quelque chose, `gdb` le garde dans l'historique pour que vous puissiez y accéder par la suite. C'est pour cela que vous voyez des "`$i` = ...", `$i` signifie qu'il s'agit de la $i^{ème}$ valeur que vous affichez. `print $1` permet d'ailleurs d'afficher `$1` (mais crée du coup une nouvelle entrée dans l'historique). Vous pouvez également afficher l'historique de la manière suivante :

— `show values n` affiche dix valeurs de l'historique, en partant de (n-5) et en allant jusqu'à (n+4).

— `print /format expr` affiche `expr` en utilisant le format spécifié. L'espace avant le "`/`" est obsolète (car une commande ne pouvant pas contenir de slash, `gdb` s'arrête de toute manière juste avant), mais il ne doit pas y en avoir après. Les formats sont les suivants :

  — `x` : entier affiché en hexadécimal

  — `d` : entier signé

  — `u` : entier non-signé

  — `o` : entier affiché en octal

  - `t` : entier affiché en binaire
  - `a` : adresse
  - `c` : caractère
  - `f` : nombre à virgule flottante (float)
- `print [/format] *adresse@taille` affiche un tableau de taille et d'adresse de départ spécifiées. Chaque élément du tableau est affiché dans le format choisi.

Nous pouvons modifier le contenu d'une variable (gdb ou de notre programme) ou même d'un registre durant l'exécution.

- `set $variable = value` permet de modifier la valeur contenue dans une variable gdb, il s'agit par exemple d'un registre (ex : "`(gdb) set $eax = 5`" pour mettre EAX à 0). Si la variable n'existe pas, elle est créée.
- `set variable variable = value` ou `set var variable = value` permet de modifier le contenu d'une variable du programme. Vous ne pouvez pas en créer de nouvelle, et vous ne pouvez pas non plus modifier la taille d'une variable.

```
(gdb) print /0 c
$3 = 2
(gdb) set var c = 14
(gdb) print c
$4 = 14
```

Chapitre 3

Opérateurs

Un opérateur est un symbole qui indique au compilateur d'exécuter une opération mathématique, relationnelle ou logique spécifique et de produire un résultat final. Les opérateurs ont une arité, une priorité et une associativité. L'arité indique le nombre d'opérandes, en C nous avons trois arités différentes :

- unaire (1 opérande)
- binaire (2 opérandes)
- ternaire (3 opérandes)

La priorité indique quels opérateurs il faut considérer en premier, c'est-à-dire quel opérateur a la priorité d'opérer sur ses opérandes. Par exemple, le langage C obéit à la convention selon laquelle la multiplication et la division ont priorité sur l'addition et la soustraction :

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int a = 2, b = 3, c = 4;
  printf("%d\n", a * b + c); // = 10
  return 0;
}
```

Si nous avons un doute la priorité peut être forcée à l'aide de parenthèses (elles ont la plus haute priorité de tous les opérateurs).
Finalement, l'associativité indique comment les opérateurs d'égale priorité sont liés par défaut. Il en existe deux types : → et de droite à gauche. Par exemple l'opérateur - réalise les opérations → :

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int a = 2, b = 3, c = 4;
  printf("%d = %d\n", a - b - c, (a - b) - c); // donne le même résultat
  return 0;
}
```

3.1 L'affectation

L'affectation, symbolisée par le signe =, est un opérateur à part entière. Son associativité est de droite à gauche et sa syntaxe est la suivante :

```
variable = expression
```

Le terme de gauche de l'affectation peut être une variable ou un élément de tableau, mais surtout pas une constante pas une constante (ne vous inquiétez pas le compilateur sera très explicite par rapport à ce sujet).

```c
#include <stdio.h>

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

```
  int a = 2; // OK
  3 = a;     // KO
  return 0;
}
```

Cette expression a pour effet d'évaluer expression et d'affecter la valeur obtenue à la variable. De plus, cette expression possède une valeur, qui est celle de l'expression.

```
#include <stdio.h>

int main(int argc, char** argv) {
  int a, b = 7;
  b = a = 4;
  printf("%d %d\n", a, b); // affiche 4 4
  return 0;
}
```

Il est important de noter que l'affectation effectue une conversion de type implicite, c'est-à-dire que la valeur de l'expression (terme de droite) est convertie dans le type du terme de gauche. Par exemple :

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i;
  float x = 2.5;
  i = 1 + x;            // i = 3 et pas 3.5
  x = x + i
  printf("\n %f \n",x); // affiche 5.5
  return 0;
}
```

3.2 Les opérateurs arithmétiques

Les opérateurs de calcul permettent de modifier mathématiquement la valeur d'une variable. Il y a un opérateur unaire, qui est le − (pour le changement de signe), et les cinq opérateurs binaires classiques que sont :

+	l'addition
−	la soustraction
*	la multiplication
/	la division
%	le modulo

Tous ces opérateurs ont une associativité de droite à gauche. Concernant l'addition, la soustraction, la division et la multiplication, ces opérateurs ont un comportement « classiques » que ce soit sur les entiers ou les flottant. En ce qui concerne le modulo, il est important de noter que cette opération ne peut être réalisé qu'entre deux entiers. Le programme suivant ne compilera pas!

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 12.5 % 3;    // KO
}
```

Pour la division, il faut faire attention au type des opérandes. En effet, si les deux opérandes sont des entiers alors c'est la division entière qui est réalisée. Sinon, si l'une des opérandes est un flottant alors c'est la division sur les flottants qui est réalisée.

```
#include <stdio.h>

int main(int argc, char** argv) {
  float x;
  x = 3 / 2;        // = 1
```

```
  x = 3 / 2.0;       // = 1.5
  return 0;
}
```

Si vous souhaitez utiliser d'autres opérateurs mathématiques vous pouvez inclure la bibliothèque `math.h` qui vous permettra d'utiliser les opérateurs de trigonométries, l'opérateur puissance, ...

3.3 Les opérateurs relationnels

Les opérateurs relationnels binaires déterminent les relations suivantes :

>	strictement supérieur
>=	supérieur ou égal
<	strictement inférieur
<=	inférieur ou égal
==	égal
!=	différent

Les opérateurs relationnels ont une associativité →. Les deux opérandes des opérateurs relationnels doivent être de type arithmétique ou pointeur. Puisqu'en C le type booléen n'existe pas, la valeur retournée est de type entier : 1 si la condition est vraie, et 0 sinon.

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 2, j = 3, k = 4;
  printf("%d %d %d\n", i < j, k <= j, i == i); // affiche 1 0 1
}
```

Attention : Il ne faut pas confondre l'opérateur d'affectation avec l'opérateur d'égalité !

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 2, j = 3, k = 4;
  printf("%d %d\n", i == j); // affiche 0
  printf("%d %d\n", i = j);  // affiche 3
}
```

3.4 Les opérateurs logiques booléens

Les opérateurs relationnels peuvent être combinés à l'aide des opérateurs logiques suivants :

&&	et logique
||	ou logique
!	négation logique

Les opérateurs && et || présentent des associativités →.
L'opérateur && a une associativité →, il retourne 0 si une des deux opérandes est à 0 et retournent 1 dans le cas contraire. Le premier opérande est complètement évalué et tous les effets secondaires sont terminés avant que l'évaluation de l'expression logique continue. Le deuxième opérande est évalué uniquement si le premier opérande est vrai, cela implique que cette évaluation élimine l'évaluation inutile du deuxième opérande lorsque le && est évalué à faux. Vous pouvez utiliser cette évaluation de court-circuit comme dans l'exemple suivant :

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i,j,k;
  // du code ...

```

```
  (i < j) && (k = i);    // k reçoit la valeur de i si i est inférieur à j
  return 0;
}
```

L'opérateur || a aussi une associativité →, il retourne 0 si les deux opérandes sont à 0 et retournent 1 dans le cas contraire. Comme pour le &&, le premier opérande doit être complètement évalué et tous les effets secondaires terminés pour que l'évaluation de l'expression se poursuive. Le deuxième opérande est évalué uniquement si le premier opérande prend la valeur 0, car l'évaluation n'est pas nécessaire lorsque l'expression est vraie. C'est aussi l'évaluation de court-circuit :

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i,j,k;
  // du code ...

  (i >= j) || (k = i);    // k reçoit la valeur de i si i est inférieur à j
}
```

L'opérateur de négation logique (!) inverse la signification de son opérande. L'opérande doit être de type arithmétique ou pointeur et il est implicitement converti en `int`. Le résultat est vrai si l'opérande converti est faux; et vice-versa. Le résultat est de type `int`.

```
#include <stdio.h>

int main(int argc, char** argv) {
  float x = 1.2, y = 0;

  printf("%d %d\n", !x, !y);     // affiche 0 et 1^
  return 0;
}
```

Bien entendu, ces opérateurs se combine pour créer des expressions logiques complexes qui pourront être utilisées comme conditions pour vos structures de contrôles (voir chapitre 4).

```
#include <stdio.h>

int main(int argc, char** argv) {
  char c;
  // du code ...

  // affiche 1 si le caractère c est une lettre de l'alphabet, et 0 sinon
  printf("%d\n", (c >= 'a' && c <= 'z') ||  (c >= 'A' && c <= 'Z'));
  return 0;
}
```

3.5 Les opérateurs logiques bit à bit

Les six opérateurs suivants permettent de manipuler des entiers au niveau du bit. Ils s'appliquent aux entiers de toute longueur (short, int ou long), signés ou non.

&	et
|	ou inclusif
∧	ou exclusif
~	complément à 1
<<	décalage à gauche
>>	décalage è droite

Les opérateurs &, | et ∧ sont des opérateurs binaires qui compare chaque bit du premier opérande au bit correspondant au deuxième opérande et calcul le résultat en fonction de la sémantique des opérateurs qui est la suivante :

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

∧	0	1
0	0	1
1	1	0

Ces opérateurs sont généralement utilisés pour masquer certaines bits. [1]

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 0;
  i = i | 0x04;
  printf("%d %d\n", i, i & 0x02);   // affiche 4 0
  printf("%d %d\n", i, i ^ 0x02);   // affiche 4 6
  return 0;
}
```

L'opérateur de complément (~), produit un complément au niveau du bit de son opérande.

~	a
1	0
0	1

L'opérande de l'opérateur de complément à un doit être un type `int`.

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int i;
  // du code ...
  printf("%d\n", i & ~077);   // met à zéro les six bits de poids faible de i
  return 0;
}
```

Les opérateurs de décalage au niveau du bit sont l'opérateur de décalage droit >>, qui déplace les bits d'une expression de type entier vers la droite, et l'opérateur de décalage gauche <<, qui déplace les bits vers la gauche.
L'opérateur de décalage gauche entraîne le décalage des bits vers la gauche par le nombre de positions spécifiées, les positions de bits qui ont été libérées par l'opération de décalage sont remplies de zéros et les bits qui sont déplacés après la fin sont ignorés, y compris le bit de signe.
L'opérateur de décalage vers la droite entraîne le décalage de bits vers la droite par le nombre de positions spécifiées. Pour les nombres non signés, les positions de bit qui ont été libérées par l'opération de décalage sont remplies de zéros. Pour les nombres signés, ... nous ne savons pas exactement ce qui sera fait. En fait cela dépendra du compilateur. Généralement, le bit de signe sera utilisé pour remplir les positions de bit libérées. Mais cela n'est pas garanti!

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 42;
  // affiche 1010, les quatre derniers bits de 42
  printf("%d%d%d%d\n", (i>>3) &1, (i>>2) &1, (i>>1) &1, i&1 );
  return 0;
}
```

Un petit exercice tiré du KR : écrivez l'expression qui inverse les `n` bits d'un entiers `x` en commençant à la position `p`.

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int n, x, p;
  // initialisation des variables n, x et p.
  x = x ^ (((1<<n) - 1) << p);
```

1. Comme exemple nous pouvons citer les options pour l'ouverture de fichier (voir chapitre 12).

```
  return 0;
}
```

3.6 Les opérateurs d'affectation composée

Les opérateurs d'assignation composée combinent l'opérateur d'assignation simple à un autre opérateur binaire. Ils exécutent l'opération spécifiée par l'opérateur supplémentaire, puis assignent le résultat à l'opérande gauche. Chacun des opérateurs + − ∗ / % >> << & ∧ | peut s'associer à l'opérateur d'affectation pour former les opérateurs suivant :

Opérateurs d'affectation composés	Opération effectuée
`expr1+=expr2`	`expr1 = expr1 + expr2`
`expr1−=expr2`	`expr1 = expr1 + expr2`
`expr1∗=expr2`	`expr1 = expr1 + expr2`
`expr1/=expr2`	`expr1 = expr1 + expr2`
`expr1%=vexpr2`	`expr1 = expr1 + expr2`
`expr1>>=expr2`	`expr1 = expr1 + expr2`
`expr1<<=expr2`	`expr1 = expr1 + expr2`
`expr1&=expr2`	`expr1 = expr1 + expr2`
`expr1∧=expr2`	`expr1 = expr1 + expr2`
`expr1|=expr2`	`expr1 = expr1 + expr2`

Il est important de noter que l'expression d'assignation composée n'équivaut pas à la version développée, car elle évalue `expr1` une seule fois, tandis que la version développée évalue `expr1` deux fois : dans l'opération d'addition et dans l'opération d'assignation.
Généralement, les opérandes d'un opérateur d'assignation composée doivent être de type entier ou flottant. En fait, il n'y a que pour += et −= où l'opérande de gauche peut être de type pointeur, auquel cas l'opérande droit doit être de type entier. Le résultat d'une opération d'assignation composée a la valeur et le type de l'opérande gauche, ce qui implique des conversions implicites.

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 0;
  i |= 0x04;
  printf("%d %d\n", i, i & 0x02);   // affiche 4 0
  printf("%d %d\n", i, i ^ 0x02);   // affiche 4 6
  return 0;
}
```

3.7 Les opérateurs d'incrémentation et de décrémentation

Les opérateurs d'incrémentation ++ et de décrémentation − s'utilisent aussi bien en suffixe qu'en préfixe. Le résultat de l'opération d'incrémentation ou de décrémentation suffixée est la valeur de l'opérande. Une fois le résultat obtenu, la valeur de l'opérande est incrémentée (ou décrémentée).

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 0, j;
  j = i++;
  printf("%d %d\n", i, j);   // affiche 1 0
  return 0;
}
```

Dans le cas de l'opération d'incrémentation ou de décrémentation préfixée la valeur de l'opérande est incrémentée (ou décrémentée) en premier lieu.

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 0, j;
  j = ++i;
  printf("%d %d\n", i, j);   // affiche 1 1
  return 0;
}
```

Notons que l'incrémentation et la décrémentation suffixées ont une priorité plus élevée que l'incrémentation et la décrémentation préfixées.

3.8 L'opérateur virgule

Une expression peut être constituée d'une suite d'expressions séparées par des virgules. Cette expression est alors évaluée → et sa valeur sera la valeur de l'expression de droite et la valeur de l'expression de droite sera perdue. Les modifications réalisées sont propagées d'expressions en expressions. Par exemple :

```c
int main(int argc, char** argv) {
  int i = 0;
  i = (i++, i++, i++);
  printf("%d %d\n", i, j);  // affiche 2
  return 0;
}
```

Puisque l'opérateur d'affectation est plus prioritaire que l'opérateur virgule, les parenthèses sont obligatoires dans le contexte qui nous intéresse ici. Ainsi, le code suivant n'aura pas le même comportement :

```c
int main(int argc, char** argv) {
  int i = 0;
  i = 1, 2, 3;
  printf("%d\n", i);  // affiche 1
  return 0;
}
```

Les virgules peuvent être utilisées comme séparateurs dans certains contextes, par exemple les listes d'arguments de fonction. Il ne faut donc pas confondre l'utilisation de la virgule comme séparateur avec son utilisation comme opérateur. En particulier l'évaluation → n'est pas garanti.

```c
int main(int argc, char** argv) {
  int i = 0;
  printf("%d %d\n", i, i++);  // affiche 1 0
  return 0;
}
```

3.9 L'opérateur conditionnel ternaire

L'opérateur conditionnel (`condition ? expr1 : expr2`) est un opérateur ternaire (il prend trois opérandes) et son fonctionnement est le suivant :

- Le premier est évalué et tous les effets secondaires sont résolus avant de continuer.
- Si le premier opérande est évalué à true (1), le deuxième opérande est évalué.
- Si le premier opérande est évalué à false (0), le troisième opérande est évalué.

Seul l'un des deux derniers opérandes est évalué dans une expression conditionnelle. Les expressions conditionnelles ont une associativité de droite à gauche. Le premier opérande doit être de type `int` ou de type pointeur.

Les règles suivantes s'appliquent au deuxième et au troisième opérandes :

- Si les deux opérandes sont du même type, le résultat est de ce type.
- Si les deux opérandes sont de types arithmétiques ou d'énumération, les conversions arithmétiques habituelles (couvertes dans Les conversions standard) sont effectuées pour les convertir en type commun.

— Si les deux opérandes sont de types même type (structure, énumération, pointeur ou `void`), le résultat prend ce type commun.
— Si l'un des opérateurs est un pointeur et le second la constante `0`, le `0` est converti en type pointeur.
— Si l'un est un pointeur sur `void`[2] et l'autre un autre pointeur, ce dernier est converti en un pointeur sur `void`.

Toutes les combinaisons des deuxième et troisième opérandes qui ne sont pas dans la liste précédente ne sont pas conformes.

```
int main(int argc, char** argv) {
  int i = 0, j = 10;
  printf("%d\n", i > j ? i : j);  // affiche 10
  return 0;
}
```

3.10 L'opérateur de conversion de type

L'opérateur de conversion de type, appelé *cast*, permet de modifier explicitement le type d'un objet. Les *casts* de types explicites sont contraints par les mêmes règles qui déterminent les effets des conversions implicites.

```
int main(int argc, char** argv) {
  int i = 1, j = 2;
  float x = i / j, y = (float)i / j;
  printf("%.1lf %.1lf\n", x, y);  // affiche : 0.0 0.5
  return 0;
}
```

D'autres restrictions sur les *casts* peuvent résulter des tailles réelles ou de la représentation de types spécifiques (nous verrons cela lorsque nous aborderons les structures au chapitre 11).

3.11 Pour résumer

Il nous reste à voir les opérateurs d'indirection et d'adresse que nous verrons au chapitre 5 lorsque nous étudierons les pointeurs.
Le tableau suivant donne la priorité et l'associativité des opérateurs en C (← pour droite-gauche et → pour gauche-droite) :

Symbole	Type d'opération	Associativité
[] () . -> ++ -- (suffixe)	Expression	→
`sizeof & * + - ! ++ --(préfixe)`	Unaire	←
casts de type	Unaire	←
* / %	Multiplicatif	→
+ -	Additive	→
≪ ≫	Décalage au niveau du bit	→
< > <= >=	Relationnel	→
== !=	Égalité	→
&	Opération de bits AND	→
∧	Opération de bits OR exclusive	→
|	Opération de bits OR inclusive	→
&&	AND logique	→
||	OR logique	→
? :	Expression-conditionnelle	←
= *= /= %= += -= <<= >>= &= ∧= |=	Affectation simple et composée	←
,	Évaluation séquentielle	→

2. Nous verrons plus en détail cela lorsque nous discuterons des pointeurs dans le chapitre 5

Chapitre 4

Les structures de contrôle

Les structures de contrôle précisent l'ordre dans lequel s'effectuent les traitements.

4.1 Les instructions et les blocs

Toute expression qui est suivie d'un point-virgule est en fait un instruction. En fait, en C le point-virgule est un terminateur d'instruction et non un séparateur.

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int a = 2;             // instruction 1
  printf("%d\n", a * 3); // instruction 2
  return 0;              // instruction 3
}
```

Les accolades { et } servent à regrouper des instructions pour obtenir instruction composée, ou bloc, qui est toujours équivalent à une instruction unique. Les accolades qui entourent les instructions de la fonction `main` en sont un exemple évident. Notons que l'accolade fermante qui termine le bloc n'est pas suivie d'un point-virgule. Il est aussi important de noter que les variables déclarées dans un bloc ne survive pas en dehors du bloc.

```c
#include <stdio.h>

int main(int argc, char** argv) {
  { int z = 4; }
  printf("%d\n", z);
  return 0;
}
```

4.2 L'instruction `if-else`

L'instruction `if-else` permet d'exprimer des prises de décision. Sa syntaxe est :

```
if (expression)
  instruction1
[else
  instruction2 ]
```

L'expression est évaluée, dans les cas où sa valeur est différent de nulle, `instruction1` s'exécute, sinon c'est `instruction2` qui s'exécute.

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 4, j = 5, k;
  if(i>j)
```

```
    k = i;
  else
    k = j;
  printf("%d\n", k);    // affiche 5
  return 0;
}
```

Comme la partie `else` est facultative, il se produit une « ambiguïté » lorsque nous considérons une imbrication de `if`. En pratique, chaque `else` s'associe avec le `if` le plus proche qui n'a pas encore de `else`.

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 4, j = 5, k = 6;
  if (i < 0)
    if (i > j)
      k = i;
    else
      k = j;
  printf("%d\n", k);  // affiche 6
  return 0;
}
```

Afin de rendre le programme plus lisible, il est préférable d'utiliser des accolades dans ce genre de situations.

4.3 L'instruction `switch`

Le `switch` est une instruction de prise de décision à choix multiples qui regarde si la valeur d'une expression entière fait partie d'une liste de constantes entières, et effectue les traitements associés à la valeur correspondante.

```
switch (expression)
{
  case constante : instructions
  ...
  case constante : instructions
  [default : instructions]
}
```

Le contrôle passe à l'instruction dont la valeur de la constante du `case` qui correspond à la valeur de expression. L'instruction `switch` peut inclure un nombre arbitraire d'instances `case`.

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 4;
  switch (i) {
    case 1:
      printf("1");
    case 2:
      printf("2");
  }
  return 0;
}
```

Toutefois, aucune valeur constant-expression dans la même `switch` ne peut avoir la même valeur. Par exemple le code suivant ne compilera pas.

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 4;
  switch (i) {
    case 1:
```

```
      printf("1");
    case 1:
      printf("1");
  }

  return 0;
}
```

L'exécution du corps de l'instruction `switch` commence à la première instruction qui correspond. L'exécution se poursuit jusqu'à la fin du corps, ou jusqu'à ce qu'une `break` instruction transfère le contrôle hors du corps.

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i = ...;

  // si i = 3 ce code affiche : 3, 4 et default.
  // si i = 1 ce code affiche : 1
  switch (i) {
    case 1:
      printf("1");
      break;
    case 2:
      printf("2");
    case 3:
      printf("3");
    case 4:
      printf("4");
    default:
      printf("default");
  }

  return 0;
}
```

L'instruction `default` est exécutée si aucune valeur n'est constante n'est égale à la valeur de expression. S'il n'y a pas d'instruction `default` et qu'aucune case correspondance n'est trouvée, aucune des instructions du `switch` corps n'est exécutée. Il peut y avoir au plus une instruction `default` et elle peut apparaître n'importe où dans le corps de l'instruction `switch`.

4.4 Les boucles – `while`, `for` et `do-while`

L'instruction while vous permet de répéter une instruction jusqu'à ce qu'une expression spécifiée devienne fausse.

```
while (expression)
  instructions
```

L'élément expression doit être de type arithmétique ou pointeur. L'exécution se déroule comme suit :

- L'élément expression est évalué.
- Si l'élément expression est initialement faux (0), le corps de l'instruction `while` pas exécuté et le contrôle passe à l'instruction suivante dans le programme.
- Si l'élément expression est vrai (différent de zéro), le corps de l'instruction est exécuté et le processus est répété à partir de l'étape 1.

Le programme suivant affiche les entiers de 0 à 9 :

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 0;

  while(i<10){
```

```
7       printf("%d\n", i++);
8     }
9     return 0;
10  }
```

L'instruction `for` est une version améliorer du `while` où il est possible de réaliser et de manière facultative une expression avant de commencer la boucle et une autre expression à la fin de chaque tour de boucle.

```
for ([expr1] ; [expr2] ; [expr3])
  instructions
```

L'exécution d'une instruction `for` se déroule comme suit :

- `expr1` est évaluée (s'il en existe une). Il n'existe aucune restriction concernant le type de cette expression.
- `expr2`, qui doit avoir un type arithmétique ou pointeur, est évaluée avant chaque itération (s'il en existe une). Trois résultats sont possibles :
  - Si `expr2` est vraie (différent de zéro), l'instruction est exécutée. À l'issue de chaque itération `expr3`. Le processus recommence ensuite avec l'évaluation de nouveau l'évaluation de `expr2`.
  - Si `expr2` est omis, alors `expr2` est considéré comme true, et l'exécution reprend exactement comme décrit dans le point précédent. Nous sommes donc dans la situation d'une boucle infinie, et il faudra utiliser une instruction à l'intérieur du corps de la boucle pour quitter le `for`.
  - Si `expr2` est fausse (0), l'exécution de l'instruction `for` se termine et le contrôle passe à l'instruction suivante dans le programme.

Les instructions `for` et `while` sont très proches. En effet, le code suivant réalise la même opération que présentée précédemment avec le `while` :

```
1  int main(int argc, char** argv) {
2    for(i = 0 ; i<10 ; i++){
3       printf("%d\n", i);
4    }
5    return 0;
6  }
```

Il est possible de simuler un `for` avec un `while` de la manière suivante :

```
epxr1;
while (expr2){
  instructions
  expr3;
}
```

En laissant vide les `expr1` et `expr2` il est possible de simuler un `while`.

```
1  #include <stdio.h>
2
3  int main(int argc, char** argv) {
4    int i = 0;
5
6    for( ; i<10 ; ){
7       printf("%d\n", i++);
8    }
9    return 0;
10 }
```

L'instruction `do-while` vous permet de répéter une instruction ou une instruction composée jusqu'à ce qu'une expression spécifiée devienne fausse.

```
do
  instructions
while (expression);
```

L'élément `expression` dans une instruction `do-while` est évalué après l'exécution du corps de la boucle. Par conséquent, le corps de la boucle est toujours exécuté au moins une fois. L'élément expression doit être de type arithmétique ou pointeur. L'exécution se déroule comme suit :

1. Le corps de l'instruction est exécuté.
2. Ensuite, l'élément expression est évalué. Si l'élément expression est faux, l'instruction `do-while` se termine et le contrôle passe à l'instruction suivante du programme. Si l'élément expression est vraie (différent de zéro), le processus se répète, en commençant à l'étape 1.

La boucle `do-while` est totalement appropriée pour gérer un menu :

```
#include <stdio.h>

int main(int argc, char** argv) {
  int rep;

  do
  {
    printf("Faire un choix :\n");
    printf("1) opération 1\n");
    printf("2) opération 2\n");
    printf("3) opération 3\n");
    printf("4) opération 4\n");
    printf("5) opération 5\n");
    scanf("%d", &rep);  // saisie utilisateur
  } while(rep != 1 && rep != 2);

  printf("Votre choix %d\n", rep);
  return 0;
}
```

Il est possible de simuler le comportement d'un `do-while` en utilisant un `while`.

```
tmp_expression = true
while(tmp_expression)
{
  instructions
  tmp_expression = expression;
}
```

Le code précédent peut donc être réécrit de la manière suivant :

```
#include <stdio.h>

int main(int argc, char** argv) {
  int rep = 42;

  while(rep != 1 && rep != 2)
  {
    printf("Faire un choix :\n");
    printf("1) opération 1\n");
    printf("2) opération 2\n");
    scanf("%d", &rep);  // saisie utilisateur
  }

  printf("Votre choix %d\n", rep);
  return 0;
}
```

4.5 Les instructions `break` et `continue`

L'instruction `break` termine l'exécution de l'instruction `do`, `for`, `switch` ou `while` englobante la plus proche dans laquelle elle figure. Le contrôle est transmis à l'instruction qui suit l'instruction terminée. Le code suivant simule le programme qui affiche les nombre entiers de 0 à 9 avec un `while`.

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 0;
```

```
5
6   while(1){
7       printf("%d\n", i++);
8       if(i>=10) break;
9   }
10  return 0;
11 }
```

L'instruction `break` est fréquemment utilisée pour mettre fin au traitement d'un cas particulier dans une instruction `switch`. L'absence d'une instruction itérative ou `switch` englobante génère une erreur.

```
1  #include <stdio.h>
2
3  int main(int argc, char** argv) {
4    int i = 4;
5    switch (i) {
6      case 1:
7        printf("1");
8        break;
9      case 2:
10       printf("2");
11       break;
12   }
13   return 0;
14 }
```

L'instruction continue passe le contrôle à l'itération suivante pour l'instruction englobante `do`, `for`, ou `while` la plus proche dans laquelle elle apparaît, en ignorant toutes les instructions restantes.
L'itération suivante pour une instruction `do`, `for`, ou `while` est déterminée comme suit :

- Dans une instruction `do` ou `while`, l'itération suivante démarre en réévaluant l'expression de l'instruction `do` ou `while`.
- Dans un `for` l'instruction `continue` provoque l'évaluation de l'expression (`expr3`) de boucle de l'instruction `for`. Ensuite, le code réévalue l'expression conditionnelle. Selon le résultat, il se termine ou itère le corps de l'instruction.

Le code suivant affiche les nombres pairs compris entre 0 et 9 :

```
1  #include <stdio.h>
2
3  int main(int argc, char** argv) {
4    int i;
5    for (i = 0; i < 10; i++){
6      if (i & 1) continue;
7      printf("%d\n", i);
8    }
9    return 0;
10 }
```

4.6 L'instruction `goto` et les étiquettes

L'instruction `goto` transfère le contrôle à une étiquette. L'étiquette donnée doit résider dans la même fonction et peut apparaître devant une seule instruction dans la même fonction. Un *label* est uniquement explicite pour une instruction `goto`. Dans tout autre contexte, une instruction étiquetée est exécutée sans tenir compte de l'étiquette. La syntaxe pour déclarer une étiquette est la suivante :

```
label:
 instructions
```

L'instruction `goto` transfère le contrôle au point étiqueté par le `label` spécifié. Le `label` doit résider dans la même fonction que le `goto` et il peut apparaître devant une seule instruction dans la même fonction.

```
goto label;
```

L'instruction goto est loin d'être en dispensable, et il est même conseillé de s'en passer. Néanmoins, il existe des situations où l'utilisation de cette instruction simplifie grandement le programme. Le plus fréquemment goto sert à sortir directement d'une structure de contrôle à plusieurs niveaux.

```
  for(...)
    ...
      for(...)
        if(grosse erreur)
          goto erreur;

erreur:
  printf("gestion de l'erreur");
```

Il est évidemment possible de se passer de cette instruction dans ce contexte, mais au prix de nombreux tests répétitifs ou d'une variable supplémentaire.
Nous pouvons aussi réécrire un `while` en utilisant un `goto` de la manière suivante :

```
1 #include <stdio.h>
2
3 int main(int argc, char** argv) {
4   int i = 0;
5 boucle:
6   printf("%d\n", i++);
7   if(i < 10) goto boucle;
8   return 0;
9 }
```

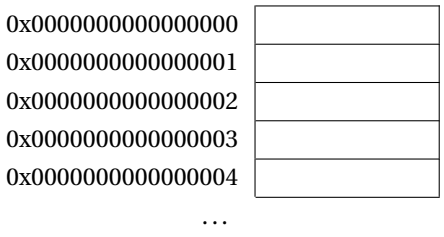
Chapitre 5

Les pointeurs

Nous pouvons assimilé la mémoire d'un programme à un grand tableau de « bits », où les variables manipulées dans un programme sont stockée. Cette mémoire est constituée d'octets qui sont identifiés de manière univoque par un numéro, que nous appelons adresse. Ainsi, il suffit de connaître l'adresse de l'octet où est stockée une variable pour la retrouver (ou, s'il s'agit d'une variable qui recouvre plusieurs octets contigus, l'adresse du premier de ces octets). Bien qu'il serait tout à fait envisageable de manipuler directement des adresses, en pratique nous identifions les variables via leurs identificateurs (nous laissons au compilateur le loisir de faire le lien entre l'identificateur et l'adresse). Néanmoins, il est parfois pratique de manipuler directement une variable par son adresse.

5.1 Adresse d'un objet

Nous avons vu jusqu'ici qu'une variable est caractérisée par trois éléments : son nom, son type et sa valeur. En fait cela n'est pas tout, car une variable est stockée quelque part dans la mémoire de l'ordinateur pendant l'exécution d'un programme. La mémoire peut être simplement vue comme un grand tableau d'octets, et chaque octet possède une adresse qui permet de l'identifier.



Lors de l'exécution d'un programme, une zone dans le mémoire va être attribuée aux variables. En fait, le nom a de la variable est utilisé pour faire référence à cette zone mémoire, qui possède une certaine adresse. Pour récupérer cette adresse nous utilisons l'opérateur `&`. Le programme `test.c` suivant sert d'illustration :

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 4, j = 1023;
  double d = 4.7;
  printf("l'adresse de i est %p et sa valeur est %d\n", &i, i);
  printf("l'adresse de j est %p et sa valeur est %d\n", &j, j);
  printf("l'adresse de d est %p et sa valeur est %lf\n", &d, d);
  return 0;
}
```

L'exécution du programme donnera :

```
$ gcc -o exec test.c
$ ./exec
l'adresse de i est 0x7fffffffe06c et sa valeur est 4
```

```
l'adresse de j est 0x7fffffffe05c et sa valeur est 1023
l'adresse de d est 0x7fffffffe060 et sa valeur est 4.700000
```

Regardons ce que nous dit l'utilitaire `gdb` sur la mémoire :

```
$ gcc -g -o exec test.c
$ gdb ./exec
...
Breakpoint 1, main (argc=1, argv=0x7fffffffe188) at hello.c:9
9   return 0;
(gdb) x/16xb 0x7fffffffe05c
0x7fffffffe05c: 0xff  0x03  0x00  0x00  0xcd  0xcc  0xcc  0xcc
0x7fffffffe064: 0xcc  0xcc  0x12  0x40  0x00  0x00  0x00  0x00
```

Nous voyons clairement apparaître les valeurs de `i` et `j`. En ce qui concerne la valeur de `d`, en fait elle apparaît au format `IEEE` et c'est donc pour cela que l'interprétation n'est pas directe. Nous pouvons aussi remarquer que les nombres ne sont pas représenter de manière « naturelle » en mémoire. En fait il existe deux types de représentation : gros-boutiste (*big-endian*) et petit-boutiste (*little-endian*). Le petit-boutiste qui est utilisé ici par le compilateur, considère en premier l'octet de poids le plus faible (contrairement au gros-boutiste qui fait l'inverse).

5.2 Notion de pointeur

Un pointeur est une variable qui contient l'adresse d'une autre variable. Pour déclarer un pointeur nous utilisons l'opérateur `*`.

```
int *p1;
int* p2;
```

L'exemple suivant illustre l'utilisation des pointeurs.

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 4;
  int* ptr_i = &i;
  printf("l'adresse de i est %p et sa valeur est %d\n", &i, i);
  printf("l'adresse de ptr_i est %p et sa valeur est %p\n", &ptr_i, ptr_i);
  return 0;
}
```

Nous voyons clairement, lors de l'exécution du code précédent, que la variable `ptr_i` contient bien l'adresse de la variable `i`.

```
l'adresse de i est 0x7fffffffe06c et sa valeur est 4
l'adresse de ptr_i est 0x7fffffffe060 et sa valeur est 0x7fffffffe06c
```

Notons qu'un pointeur est une variable comme une autre, et la donnée qu'elle peut contenir est une adresse d'une zone mémoire. Une variable de type pointeur occupe donc également de la place en mémoire, plus exactement 8 octets sur une architecture 64 bits.

```
Breakpoint 1, main (argc=1, argv=0x7fffffffe188) at hello.c:8
8   return 0;
(gdb) x/16xb 0x7fffffffe060
0x7fffffffe060: 0x6c  0xe0  0xff  0xff  0xff  0x7f  0x00  0x00
0x7fffffffe068: 0x00  0x00  0x00  0x00  0x04  0x00  0x00  0x00
```

Ce n'est pas parce qu'un pointeur sert à représenter un adresse qu'il doit contenir une adresse valide. Par exemple le code suivant est correct (le compilateur va tousser un peu ...) :

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 4;
  int* ptr_i = 1664;
  printf("l'adresse de i est %p et sa valeur est %d\n", &i, i);
```

```
  printf("l'adresse de ptr_i est %p et sa valeur est %p\n", &ptr_i, ptr_i);
  return 0;
}
```

Bien que le code précédent soit correct syntaxiquement, il est incorrect sémantiquement. En effet, il est possible de consulter la valeur qui se trouve dans la mémoire à une certaine adresse en utilisant l'opérateur de déréférencement. Cet opérateur, noté *, est à utiliser suivi d'une variable de type pointeur et renvoie la valeur se trouvant à l'adresse stockée dans le pointeur. Le code suivant illustre l'utilisation de cet opérateur :

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 4;
  int* ptr_i = &i;
  printf("ptr_i pointe sur l'adresse %p, dont la valeur pointée est %d\n", ptr_i, *ptr_i);
  return 0;
}
```

Le résultat du programme précédent est bien celui attendu :

```
ptr_i pointe sur l'adresse 0x7ffe523c1054, dont la valeur pointée est 4
```

Lorsque l'adresse pointée est incorrect, comme illustré dans le programme suivant, alors vous obtiendrez une erreur de segmentation.

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 4;
  int* ptr_i = 1664;
  printf("ptr_i pointe sur l'adresse %p, dont la valeur pointée est %d\n", ptr_i, *ptr_i);
  return 0;
}
```

L'erreur de segmentation est une erreur assez commune, et il est donc important de savoir identifier et fixer votre programme de manière à ce que cette erreur n'arrive plus. Pour cela vous pouvez utiliser les utilitaires `gdb` et `valgrind`. `gdb` vous permettra d'obtenir le numéro de ligne où votre programme crash.

```
(gdb) run
Starting program: /home/lagniez/Works/Enseignement/c-av-c/cours/exemples/exec
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
l'adresse de i est 0x7fffffffe064 et sa valeur est 4

Program received signal SIGSEGV, Segmentation fault.
0x0000555555555179 in main (argc=1, argv=0x7fffffffe188) at hello.c:7
7   printf("ptr_i pointe sur l'adresse %p, dont la valeur pointée est %d\n",
(gdb) bt
#0  0x0000555555555179 in main (argc=1, argv=0x7fffffffe188) at hello.c:7
```

Quant à `valgrind` il sera plus précis (de manière générale il est plus approprié d'utiliser `valgrind` pour les erreurs concernant la mémoire).

```
$ valgrind ./exec
==22090== Memcheck, a memory error detector
==22090== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==22090== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==22090== Command: ./exec
==22090==
==22090== Invalid read of size 4
==22090==    at 0x109179: main (hello.c:7)
==22090==  Address 0xa4 is not stack'd, malloc'd or (recently) free'd
==22090==
==22090==
==22090== Process terminating with default action of signal 11 (SIGSEGV)
==22090==  Access not within mapped region at address 0xA4
```

```
==22090==    at 0x109179: main (hello.c:7)
==22090==  If you believe this happened as a result of a stack
==22090==  overflow in your program's main thread (unlikely but
==22090==  possible), you can try to increase the size of the
==22090==  main thread stack using the --main-stacksize= flag.
==22090==  The main thread stack size used in this run was 8388608.
==22090==
==22090== HEAP SUMMARY:
==22090==     in use at exit: 1,024 bytes in 1 blocks
==22090==   total heap usage: 1 allocs, 0 frees, 1,024 bytes allocated
==22090==
==22090== LEAK SUMMARY:
==22090==    definitely lost: 0 bytes in 0 blocks
==22090==    indirectly lost: 0 bytes in 0 blocks
==22090==      possibly lost: 0 bytes in 0 blocks
==22090==    still reachable: 1,024 bytes in 1 blocks
==22090==         suppressed: 0 bytes in 0 blocks
==22090== Rerun with --leak-check=full to see details of leaked memory
==22090==
==22090== For lists of detected and suppressed errors, rerun with: -s
==22090== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Erreur de segmentation
```

Un pointeur étant également une variable, et il possède donc aussi une adresse. Nous pouvons donc stocker cette adresse dans un pointeur. Puisqu'il s'agit d'une variable pointeur, nous ajoutons donc une `*`, et l'adresse qui y sera stockée sera celle d'une variable de type pointeur.

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 4;
  int* ptr_i = &i;
  int** ptr_ptr_i = &ptr_i;
  printf("%d %d %d\n", i, *ptr_i, **ptr_ptr_i);
  return 0;
}
```

Il y a deux pointeurs particuliers que nous pouvons utiliser dans certaines situations spécifiques. Le pointeur `NULL` est celui qui ne pointe vers aucune zone mémoire. Plus précisément, `NULL` représente l'adresse `0x00`, mais puisque cette adresse ne peut pas être associée à une variable cela ne pose pas de problème.

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int* ptr = NULL;
  return 0;
}
```

Il y a aussi le pointeur générique, noté `void*`, qui est un type particulier de pointeur qui permet de stocker un pointeur vers n'importe quel type de donnée. Il peut donc être converti vers n'importe quel autre type de pointeur, et ces derniers peuvent être convertis vers lui.

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 4;
  void* ptr = &i;
  return 0;
}
```

En fait, les pointeurs génériques sont très utiles dès que nous souhaitons implémenter la généricité. Dans ce cas, il est possible en connaissant le type de l'objet pointé et d'utiliser l'opérateur de cast de récupérer la valeur pointée.

```c
#include <stdio.h>

```

```
int main(int argc, char** argv) {
  int i = 4;
  void* ptr = &i;
  printf("%d\n", *((int *)ptr));
  return 0;
}
```

En fait le C est un langage assez permissif et il est toujours possible de forcer le cast. Par exemple le code suivant est correct (le compilateur tousse un peu ...) :

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i = 4;
  float *ptr = &i;
  printf("%d\n", *((int *)ptr));
  return 0;
}
```

Chapitre 6

Les fonctions

Le langage C est un langage modulaire, c'est-à-dire qu'un ensemble d'actions élémentaires ayant une certaine cohérence peuvent être regroupées dans une fonction, créant ainsi de nouveaux types d'action, utilisables simplement par leur nom. Une seule de ces fonctions existe obligatoirement : c'est la fonction principale `main`. Dans le `main` il est possible d'appeler une ou plusieurs fonctions secondaires. De même, chaque fonction secondaire peut appeler d'autres fonctions secondaires ou s'appeler elle-même (récursivité).

6.1 Définition et appel d'une fonction

Une fonction est définie par son type de retour, ses paramètres et la donnée du texte de son algorithme, appelé corps de la fonction.

```
type nom_fonction(param)
{
   liste d'instructions
}
```

La première ligne de cette définition est l'en-tête de la fonction. Dans cet en-tête, type désigne le type de retour de la fonction. Une fonction qui ne renvoie pas de valeur est une fonction dont le type est spécifié par le mot-clef `void`. `param` représente les paramètres formels de la fonction (par opposition aux paramètres effectifs avec lesquels la fonction est appelée). Les paramètres formels peuvent être de n'importe quel type. Leurs identificateurs n'ont d'importance qu'à l'intérieur de la fonction. La fonction peut ne pas ne posséder de paramètres, dans ce cas il est possible de remplacer la liste de paramètres formels par le mot-clef `void` ou de ne rien mettre. Considérons comme exemple concret :

```
1  #include <stdio.h>
2
3  int addInt(int a, int b)
4  {
5    printf("Pour le moment je ne fais rien\n");
6  }
7
8  int main(int argc, char ** argv)
9  {
10   return 0;
11 }
```

Dans cet exemple, la fonction `addInt` retourne une variable de type `int`. Elle prend en paramètre deux variables de types `int`. Dans le corps de la fonction elle affiche un petit message pour nous dire qu'elle ne fait rien.
Bien que la fonction est censée retourner un objet de type `int`, nous pouvons voir que notre fonction ne retourne rien. En fait cela est une possibilité, et le compilateur s'en portera très bien. Cependant, de manière générale lorsque nous déclarons qu'une fonction retourne un objet d'un certain type c'est pour

réaliser un certain nombres d'opérations permettant de calculer un résultat que nous souhaitons récupérer. Pour réaliser cela, il suffit d'utiliser le mot-clef `return`.

```
#include <stdio.h>

int addInt(int a, int b)
{
  return a + b;
}

int main(int argc, char ** argv)
{
  return 0;
}
```

La valeur de retour de la fonction doit être du même que celui qui a été spécifié dans l'en-tête de la fonction. Si la fonction ne retourne pas de valeur, il est possible d'utiliser l'instruction `return` sans spécifier d'expression.

```
#include <stdio.h>

void sleep()
{
  return;
}

int main(int argc, char ** argv)
{
  return 0;
}
```

Dès que l'instruction `return` est rencontrée, la fonction est stoppée et un retour au programme appelant sera alors provoqué. Ainsi, plusieurs instructions `return` peuvent apparaître dans une fonction et c'est le premier `return` rencontré lors de l'exécution qui stoppera la fonction.

```
#include <stdio.h>

int pouvoirConduire(int age)
{
  if(age >= 18) return 1;
  return 0;
}

int main(int argc, char ** argv)
{
  printf("ok : %d\n", age(22));
  return 0;
}
```

L'appel d'une fonction se fait en utilisant le nom de la fonction et en spécifiant ses paramètres. L'ordre et le type des paramètres effectifs de la fonction doivent concorder avec les paramètres formels de la fonction. Les paramètres effectifs peuvent être des expressions. Comme spécifié dans la section 3.8, la virgule qui sépare deux paramètres effectifs est un simple signe de ponctuation (donc attention à l'utilisation des opérations d'incrémentation et de décrémentation).

6.2 Durée de vie et portée des variables

Nous distinguons deux catégories de variables en C en fonction de leur durée de vie.

6.2.1 Variables globales

Une variable globale est une variable déclarée en dehors de toute fonction. Elles sont systématiquement permanentes et connues du compilateur dans toute la portion de code qui suit sa déclaration. De plus,

elles sont initialisées à zéro lors de la déclaration.

```c
#include <stdio.h>

int n;

void a() { printf("a aime n (%d)\n", ++n); }

void b() { printf("b aime n (%d)\n", ++n); }

int main(int argc, char** argv) {
  printf("main aime n (%d)\n", ++n);
  a();
  b();
  return 0;
}
```

6.2.2 Variables locales

Une variable locale est une variable déclarée à l'intérieur d'une fonction (ou d'un bloc d'instructions) du programme. Ce sont par défaut des variables temporaires, c'est-à-dire que leurs emplacement en mémoire est alloué sur la pile de façon dynamique lors de l'exécution du programme. Ces variables ne sont pas initialisées par défaut et leur emplacement en mémoire est libéré à la fin de l'exécution de la fonction. Dans ce cas, la variable est dite automatique (il est possible d'utiliser le spécificateur de type correspondant, `auto`, mais il ne s'applique qu'aux variables temporaires qui sont automatiques par défaut).
Une variable temporaire peut également être placée dans un registre de la machine. Les registres sont plus rapides à accéder que la mémoire, de sorte qu'il est possible de demander au compilateur de placer les variables les plus fréquemment utilisées peuvent être placées dans des registres à l'aide du mot clé de `register`. Néanmoins, le nombre de registres étant limité, cette requête ne sera satisfaite que s'il reste des registres disponibles. De plus, grâce aux performances des optimiseurs de code intégrés au compilateur, cette technique permettant d'accélérer les programmes a aujourd'hui perdu tout son intérêt.
La durée de vie des variables est liée à leur portée, c'est-à-dire à la portion du programme dans laquelle elles sont définies.

```c
#include <stdio.h>

void compte) {
  int i = 0;
  while(i < 10) printf("%d\n", i++);
}

int main(int argc, char** argv) {
  compte();   // compte de 0 à 9
  compte();   // compte de 0 à 9
  return 0;
}
```

En fait, ce qui va définir la portée d'une variable est le bloc dans lequel elle apparaît. Par exemple le code suivant est correct :

```c
#include <stdio.h>
void portee() {
  int cpt = 4;
  { printf("%d\n", cpt); }
}

int main(int argc, char** argv) {
  portee();
  return 0;
}
```

Tandis que le code suivant ne compilera pas!

```c
#include <stdio.h>
void portee() {
  { int cpt = 4; }
  printf("%d\n", cpt);
}

int main(int argc, char** argv) {
  portee();
  return 0;
}
```

Notons que les variables locales n'ont en particulier aucun lien avec des variables globales de même nom.

```c
#include <stdio.h>

int n;

void a() { printf("a aime n (%d)\n", ++n); }

void b() {
  int n = 0;
  n++;
  printf("%d\n", n);
}

int main(int argc, char** argv) {
  b();
  a();
  return 0;
}
```

Les variables locales à une fonction ont une durée de vie limitée à une seule exécution de cette fonction. Leurs valeurs ne sont pas conservées d'un appel au suivant. Une variable permanente est caractérisée par le mot-clef `static`. Elle occupe un emplacement en mémoire qui est défini une fois pour toutes lors de la compilation et qui reste le même durant toute l'exécution du programme. Par défaut, les variables permanentes sont initialisées à zéro par le compilateur.

```c
#include <stdio.h>

void compteur()
{
  static int cpt;
  printf("tu m'as appelé %d fois\n", ++cpt);
}

int main(int argc, char ** argv)
{
  compteur();
  compteur();
  return 0;
}
```

6.3 Paramètres d'une fonction

Les paramètres d'une fonction sont traités de la même manière que les variables locales. Plus précisément, lors de l'appel de la fonction, les paramètres effectifs sont copiés dans le segment de pile. La fonction travaille alors uniquement sur cette copie, qui disparaît lors du retour au programme appelant. Les paramètres sont donc passés par valeurs. Cela implique en particulier que, si la fonction modifie la valeur d'un de ses paramètres, seule la copie sera modifiée. Considérons l'exemple suivant n'a pas le comportement attendu :

```c
#include <stdio.h>

```

```c
void swap(int a, int b) {
  int tmp = a;
  printf("avant swap : a = %d et b = %d\n", a, b);
  a = b;
  b = tmp;
  printf("après swap : a = %d et b = %d\n", a, b);
}

int main(int argc, char** argv) {
  int a = 4, b = 2;
  swap(a, b);
  printf("dans main : a = %d et b = %d\n", a, b);
  return 0;
}
```

Néanmoins, il reste possible de modifier la valeur d'une variable dans le contexte d'une fonction en passant son adresse mémoire. Par exemple, pour échanger les valeurs de deux variables, il faut écrire :

```c
#include <stdio.h>

void swap(int *a, int *b) {
  int tmp = *a;
  printf("avant swap : a = %d et b = %d\n", *a, *b);
  *a = *b;
  *b = tmp;
  printf("après swap : a = %d et b = %d\n", *a, *b);
}

int main(int argc, char **argv) {
  int a = 4, b = 2;
  swap(&a, &b);
  printf("dans main : a = %d et b = %d\n", a, b);
  return 0;
}
```

6.4 Passage de paramètres à la fonction `main`

Parmi toutes les fonctions qui vont constituer votre programme, la fonction `main` est une peu particulière. Elle constitue le point d'entrée de votre programme et elle permet la communication entre le système d'exploitation et votre programme. Il faut noter, que le `main` peut être écrit de quatre manières différentes en fonction de ce que vous voulez échanger avec le système d'exploitation. Voici les quatre façons d'écrire la fonction main. [1]

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

Pour être précis la fonction `main` est de type `int`. Elle doit donc retourner un entier dont la valeur est transmise à l'environnement d'exécution. C'est pour cela que ce programme compilé avec l'option `-Wall` affiche un *warning*.

```c
#include <stdio.h>

void main(int argc, char *argv []) {}
```

```
$ gcc -o exec hello.c -Wall
hello.c:3:6: warning: return type of 'main' is not 'int' [-Wmain]
    3 | void main(int argc, char* argv[]) {}
      |      ^~~~
```

1. Jusqu'ici nous avons utilisé `char **argv` à la place de `char *argv[]`. Cela ne pose pas de problème car nous verrons dans le chapitre 7 que nous pouvons caractériser un tableau par un pointeur (et surtout cela me fait économiser un caractère).

Cet entier indique si le programme s'est ou non déroulé sans erreur. Contrairement au C, la valeur de retour 0 correspond à une terminaison correcte, toute valeur de retour non nulle correspond à une terminaison sur une erreur. Il existe deux constantes symboliques `EXIT SUCCESS` (égale à 0) et `EXIT FAILURE` (égale à 1) définies dans `stdlib.h`. L'instruction return(statut) dans la fonction main peut être remplacée par un appel à la fonction `exit` de la librairie standard (`stdlib.h`).

```
int main(int argc, char ** argv)
{
  printf("Hello World!"));
  exit(0);
}
```

Un programme C peut recevoir une liste d'arguments venant de la ligne de commande qui sert à lancer le programme. La fonction main reçoit tous ces éléments de la part de l'interpréteur de commandes. Pour cela, nous utilisons la version du `main` qui possède deux paramètres formels. `argc` est une variable de type int qui donne le nombre de mots composant la ligne de commande (y compris le nom de l'exécutable). Elle est donc égale au nombre de paramètres effectifs de la fonction + 1. `argv` est un tableau de chaînes de caractères correspondant chacune à un mot de la ligne de commande (`argv[0]` contient donc le nom de l'exécutable). Par exemple, le programme suivant affiche les paramètres de la ligne de commande.

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  int i;
  for (i = 0; i < argc; i++) printf("%s\n", argv[i]);
  return 0;
}
```

Nous lançons le programme de la manière suivante :

```
$ gcc -o exec hello.c
$ ./exec 1 2 3 bonjour toto
1
2
3
bonjour
toto
```

6.5 *Forward* déclaration

Il existe des situations où nous avons besoin d'utiliser une fonction avant que cette dernière ne soit déclarée. Cela arrive par exemple lorsque nous souhaitons utiliser de la récursivité croisée.

```
#include <stdio.h>

int pair(int n) {
  if (n==0) return 1;
  return impair(n-1);
}

int impair(int n) {
  if (n==0) return 0;
  return pair(n-1);
}

int main(int argc, char ** argv)
{
  printf("pair %d\n", pair(33));
  return 0;
}
```

Le code précèdent, bien que correct algorithmiquement, ne compilera pas!

```
$ gcc -g -o exec hello.c
hello.c: In function 'pair':
hello.c:5:10: warning: implicit declaration of function 'impair'; did you mean 'pair'? [-Wimplicit-
    function-declaration]
    5 |   return impair(n - 1);
      |          ^~~~~~
      |          pair
```

En fait, le compilateur à besoin d'avoir l'information que la fonction existe pour achever la compilation. Afin de pallier ce problème, il est possible de déclarer une fonction à l'aide d'un prototype. Celui-ci permet de spécifier le type de retour de la fonction, son nombre d'arguments et leur type, mais ne comporte pas le corps de cette fonction (le compilateur retrouvera ces petits à l'édition de liens).

```c
#include <stdio.h>

int impair(int);

int pair(int n) {
  if (n==0) return 1;
  return impair(n-1);
}

int impair(int n) {
  if (n==0) return 0;
  return pair(n-1);
}

int main(int argc, char ** argv)
{
  printf("pair %d\n", pair(33));
  return 0;
}
```

6.6 Pointeur sur une fonction

Même les fonctions ont une adresse mémoire, il est donc possible d'avoir un pointeur qui pointe sur une fonction. Cela est parfois utile, pour passer une fonction comme paramètre d'une autre fonction et donc de faire de la généricité. Un pointeur sur une fonction correspond à l'adresse du début du code de la fonction. Pour déclarer un pointeur de fonction, il suffit de considérer les fonctions comme des variables. Leur déclaration est la suivante :

```c
type (*ptr) (paramètres);
```

Voici un exemple d'utilisation concret (mais un peu bidon) :

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

void test(int a) { printf("test %d\n", a); }

int main(int argc, char* argv[]) {
  void (*ptr)(int) = &test;
  ptr(10);
  return 0;
}
```

6.7 Fonctions avec un nombre variable de paramètres

Il est possible de définir des fonctions dont le nombre de paramètres est variable. En fait, vous en avez déjà utilisé une plusieurs fois : la fonction `printf`. Jetons un œil à la page `man` de la fonction `printf`.

```
$ man 3 printf
PRINTF(3)                                                    Linux Programmer's Manual

                                                                          PRINTF(3)

NAME
       printf, fprintf, dprintf, sprintf, snprintf, vprintf, vfprintf, vdprintf, vsprintf, vsnprintf -
      formatted output conversion

SYNOPSIS
       #include <stdio.h>

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

Une fonction possédant un nombre variable de paramètre doit posséder au moins un paramètre formel fixe. Pour spécifier que le nombre d'arguments est variable nous utilisons la notation `...` qui est obligatoirement à la fin de la liste de paramètre. Cela spécifie que la fonction possède un nombre quelconque de paramètres (éventuellement de types différents) en plus des paramètres formels fixes.
Un appel à une fonction ayant un nombre variable de paramètres s'effectue comme un appel à n'importe quelle autre fonction. Pour accéder à la liste des paramètres, nous utilisons des macros définies dans le fichier `stdarg.h` de la librairie standard. Pour récupérer les arguments il faut déclarer dans le corps de la fonction une variable pointant sur la liste des paramètres de l'appel : cette variable a pour type `va_list`.

```
va_list args;
```

Cette variable est alors initialisée avec la macro `va_start` (voir la page `man`).

```
void va_start(va_list ap, last);
```

La liste est libérée via un appel à la macro `va_end`.

```
void va_end(va_list ap);
```

Pour accéder aux différents paramètres de liste nous utilisons la macro `va_arg` qui retourne le paramètre suivant de la liste.

```
type va_arg(va_list ap, type);
```

Notons qu'il n'est pas possible de récupérer le nombre de paramètre de la liste. Pour cela, nous utilisons généralement un paramètre formel qui correspond au nombre de paramètres de la liste, ou une valeur particulière qui indique la fin de la liste.

```
1  #include <stdarg.h>
2  #include <stdio.h>
3
4  void varArg(int nb, ...) {
5    int v;
6    va_list args;
7
8    va_start(args, nb);
9    while (nb--) {
10     v = va_arg(args, int);
11     printf("%d\n", v);
12   }
13   va_end(args);
14 }
15
16 int main(int argc, char* argv[]) {
17   varArg(2, 1, 2);
18   varArg(3, 1, 2, 3);
19   return 0;
20 }
```

Chapitre 7

Les tableaux

Une déclaration de tableau nomme le tableau et spécifie le type de ses éléments. C'est un ensemble fini d'éléments de même type, stockés en mémoire de manière contiguë. Il est possible de définir le nombre d'éléments du tableau de manière explicite ou implicite. Une variable de type tableau est considérée comme un pointeur vers un gros bloc mémoire qui représente les éléments du tableau.

7.1 Tableaux à une dimension

La définition d'un tableau nécessite trois informations :

1. le type des éléments du tableau
2. le nom du tableau
3. la longueur du tableau

La longueur d'un tableau peut être définie explicitement ou implicitement. Dans le cas de la déclaration explicite la syntaxe est la suivante :

```
type id[longueur];
```

Nous pouvons par exemple écrire un programme, stocké dans `test.c`, qui déclare un tableau d'entiers de taille 5 :

```
1 #include <stdio.h>
2
3 int main(int argc, char ** argv)
4 {
5   int tab[5];
6   return 0;
7 }
```

Essayons de voir ce que ce tableau contient en utilisant l'utilitaire `gdb` :

```
$ gcc -g -o exec test.c
$ gdb ./exec
...
(gdb) break main
Breakpoint 1 at 0x1134: file hello.c, line 4.
(gdb) run
Starting program: /home/lagniez/Works/Enseignement/c-av-c/cours/exemples/exec
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0x7fffffffe188) at hello.c:5
5   return 0;
(gdb) print tab
$1 = {0, 0, 64, 0, 16}
```

Nous pouvons voir que le tableau comporte 5 entiers, mais que certaines valeurs sont différentes de 0 (contrairement à `java`). En fait, en C il n'y a que lorsque le tableau est `global` ou `static` [1] que tous ses éléments sont initialisés à zéro. Néanmoins, depuis la norme `C23`, il existe une méthode d'initialisation permettant de remplir le tableau de zéros.

```c
#include <stdio.h>

int main(int argc, char ** argv)
{
  int tab[5] = {};
  return 0;
}
```

Voyons ce que nous dit `gdb` :

```
Breakpoint 1, main (argc=1, argv=0x7fffffffe188) at hello.c:4
4   int tab[5] = {0};
(gdb) print tab
$1 = {0, 0, 64, 0, 16}
(gdb) next
5   return 0;
(gdb) print tab
$2 = {0, 0, 0, 0, 0}
```

Il est possible d'initialiser un tableau de manière séquentielle de la manière suivante :

```c
#include <stdio.h>

int main(int argc, char ** argv)
{
  int tab[5] = {0, 1, 2, 3, 4};
  return 0;
}
```

Dans ce cas, puisque la taille peut être calculée à la compilation, cette dernière n'est pas nécessaire dans la déclaration de la variable. Ainsi le code suivant a exactement le même comportement que le précédent :

```c
#include <stdio.h>

int main(int argc, char ** argv)
{
  int tab[] = {0, 1, 2, 3, 4};
  return 0;
}
```

Regardons un peu le contenu de la mémoire en utilisant `gdb` :

```
Breakpoint 1, main (argc=1, argv=0x7fffffffe188) at hello.c:9
9   struct complex tab[] = {0, 1, 2, 3, 4};
(gdb) next
11    return 0;
(gdb) print tab
$1 = {0, 1, 2, 3, 4}
```

Pour le moment nous avons choisi d'initialiser nos tableaux avec des types simples, mais il est tout à fait possible d'utiliser des types composés.

```c
#include <stdio.h>

struct complex {
  float r;
  float im;
};

int main(int argc, char** argv) {
  struct complex tab[] = {{0, 1}, {1, 1}};
  return 0;
}
```

1. Nous verrons ces notions en détail lorsque nous aborderons la programmation modulaire au chapitre 14.

Qu'est ce que nous raconte gdb sur ce morceau de code :

```
Breakpoint 1, main (argc=1, argv=0x7fffffffe188) at hello.c:9
9   struct complex tab[] = {{0, 1}, {1, 1}};
(gdb) next
11    return 0;
(gdb) print tab
$1 = {{r = 0, im = 1}, {r = 1, im = 1}}
```

Il est par ailleurs possible de désigner spécifiquement les éléments du tableau que vous souhaitez initialiser. Cette initialisation sélective est réalisée à l'aide du numéro du ou des éléments de la manière suivante :

```
#include <stdio.h>

int main(int argc, char** argv) {
  int tab[5] = {[2] = 42, [3] = 1664};
  return 0;
}
```

Nous pouvons vérifier avec gdb le contenu du tableau.

```
Breakpoint 1, main (argc=1, argv=0x7fffffffe188) at hello.c:4
4   int tab[] = {[2] = 42, [3] = 1664};
(gdb) next
5   return 0;
(gdb) print tab
$1 = {0, 0, 42, 1664}
```

Depuis la norme C99, il est aussi possible de définir des plages de cases du tableau que nous souhaitons assigner. La syntaxe est la suivante :

```
#include <stdio.h>

int main(int argc, char** argv) {
  int tab[5] = {[0 ... 2] = -1, [3 ... 4] = 1};
  return 0;
}
```

Nous obtenons bien ce que nous recherchons, c'est-à-dire que les trois premières cases du tableau son assignées à -1 et les deux dernières à 1.

```
Breakpoint 1, main (argc=1, argv=0x7fffffffe188) at hello.c:4
4   int tab[5] = {[0 ... 2] = -1, [3 ... 4] = 1};
(gdb) next
5   return 0;
(gdb) print tab
$1 = {-1, -1, -1, 1, 1}
```

Quelque soit l'approche sélective utilisée, il n'est pas nécessaire de spécifier la longueur du tableau à l'initialisation. Ainsi le code suivante est correct :

```
#include <stdio.h>

int main(int argc, char** argv) {
  int tab[] = {[0 ... 2] = -1, [3 ... 4] = 1, [10] = 1000};
  return 0;
}
```

Ce qui donne le résultat suivant :

```
Breakpoint 1, main (argc=1, argv=0x7fffffffe188) at hello.c:4
4   int tab[] = {[0 ... 2] = -1, [3 ... 4] = 1, [10] = 1000};
(gdb) next
5   return 0;
(gdb) print tab
$1 = {-1, -1, -1, 1, 1, 0, 0, 0, 0, 0, 1000}
```

Attention : si la longueur du tableau est spécifiée, alors il est impérative que l'assignation soit en adéquation avec cette dernière. Par exemple le code suivante ne compile pas :

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int tab[5] = {[0 ... 2] = -1, [3 ... 4] = 1, [10] = 1000};
  return 0;
}
```

Toutes ces initialisations se combinent. Par exemple le code suivante est correct :

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int tab[] = {0,1, [4] = 3, [6 ... 10] = 42};
  return 0;
}
```

Le résultat observé à l'aide de gdb est bien celui attendu :

```
Breakpoint 1, main (argc=1, argv=0x7fffffffe188) at hello.c:4
4   int tab[] = {0, 1, [4] = 3, [6 ... 10] = 42};
(gdb) next
5   return 0;
(gdb) print tab
$1 = {0, 1, 0, 0, 3, 0, 42, 42, 42, 42, 42}
```

L'accès aux éléments d'un tableau se réalise à l'aide de la notation crochet et de l'un indice exprimé avec un entier. Ce nombre entier correspondant à la position de chaque élément dans le tableau. Soulignons, que les indices commencent toujours à zéro. Le code suivant parcours un tableau et affiche ses éléments sur la sortie standard :

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int i;
  int tab[] = {0,1, [4] = 3, [6 ... 10] = 42};
  for(i = 0 ; i<11 ; i++) printf("%d\n", tab[i]);
  return 0;
}
```

Dans le cas où une expression de type tableau est fournie comme opérande de l'opérateur `sizeof`, alors le résultat de celui-ci sera le nombre d'octets totale du tableau. [2]

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int tab[] = {0,1, [4] = 3, [6 ... 10] = 42};
  printf("%d\n", sizeof(tab));   // affiche 44
  return 0;
}
```

Dans ce cas il est tout à fait possible de récupérer la taille du tableau à l'exécution de la manière suivante :

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int tab[] = {0,1, [4] = 3, [6 ... 10] = 42};
  printf("%d %d\n", sizeof(tab), sizeof(tab) / sizeof(tab[0]));   // affiche 44 et 11
  return 0;
}
```

Une des erreurs les plus fréquentes consiste à dépasser la taille d'un tableau, c'est-à-dire tenter d'accéder à un objet qui ne fait pas partie de votre tableau.

```c
#include <stdio.h>

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

2. Nous verrons plus loin lorsque que cela n'est pas vrai sur les pointeurs (voir chapitre 5), même si ce dernier pointe vers un tableau.

```
4   int i;
5   int tab[] = {0, 1, [4] = 3};
6   for (i = 0; i < 10; i++) printf("%d ", tab[i]);
7   printf("\n");
8   return 0;
9 }
```

Comme nous pouvons le voir ce code ne crashera pas systématiquement :

```
$ gcc -g -o exec test.c
$ ./exec
0 1 0 0 3 0 0 7 1 0
```

Ainsi que nous pouvons l'observer même l'utilitaire `gdb` ne se rendra pas compte de l'erreur :

```
0 1 0 0 3 0 0 7 1 0
[Inferior 1 (process 32664) exited normally]
```

Pour éviter ce genre de situation, il est possible d'utiliser un autre utilitaire : `valgrind`. `valgrind` est un outil de programmation libre pour déboguer, effectuer du profilage de code et mettre en évidence des fuites mémoires. Par exemple lorsque nous exécutons `valgrind` sur notre code (compilé avec l'option `-g`), `valgrind` est capable de se rendre compte que vous avez essayé d'accéder à une zone mémoire qui ne vous appartient pas.

```
$ valgrind ./exec
==32850== Memcheck, a memory error detector
==32850== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==32850== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==32850== Command: ./exec
==32850==
==32850== Conditional jump or move depends on uninitialised value(s)
==32850==    at 0x48D30B7: __vfprintf_internal (vfprintf-internal.c:1646)
==32850==    by 0x48BF42A: printf (printf.c:33)
==32850==    by 0x10919A: main (hello.c:6)
==32850==
==32850== Use of uninitialised value of size 8
==32850==    at 0x48B960B: _itoa_word (_itoa.c:179)
==32850==    by 0x48D2CCF: __vfprintf_internal (vfprintf-internal.c:1646)
==32850==    by 0x48BF42A: printf (printf.c:33)
==32850==    by 0x10919A: main (hello.c:6)
==32850==
==32850== Conditional jump or move depends on uninitialised value(s)
==32850==    at 0x48B961C: _itoa_word (_itoa.c:179)
==32850==    by 0x48D2CCF: __vfprintf_internal (vfprintf-internal.c:1646)
==32850==    by 0x48BF42A: printf (printf.c:33)
==32850==    by 0x10919A: main (hello.c:6)
==32850==
==32850== Conditional jump or move depends on uninitialised value(s)
==32850==    at 0x48D3583: __vfprintf_internal (vfprintf-internal.c:1646)
==32850==    by 0x48BF42A: printf (printf.c:33)
==32850==    by 0x10919A: main (hello.c:6)
==32850==
==32850== Conditional jump or move depends on uninitialised value(s)
==32850==    at 0x48D2DEB: __vfprintf_internal (vfprintf-internal.c:1646)
==32850==    by 0x48BF42A: printf (printf.c:33)
==32850==    by 0x10919A: main (hello.c:6)
==32850==
0 1 0 0 3 0 0 7 1 0
==32850==
==32850== HEAP SUMMARY:
==32850==     in use at exit: 0 bytes in 0 blocks
==32850==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==32850==
==32850== All heap blocks were freed -- no leaks are possible
==32850==
==32850== Use --track-origins=yes to see where uninitialised values come from
```

```
==32850== For lists of detected and suppressed errors, rerun with: -s
==32850== ERROR SUMMARY: 10 errors from 5 contexts (suppressed: 0 from 0)
```

Nous verrons d'autres aspects de l'utilitaire `valgrind` lorsque nous aborderons l'allocation dynamique de la mémoire.

7.2 Tableaux multidimensionnels

Le langage C vous permet de créer et de gérer des tableaux dit multidimensionnels (en fait, des tableaux de tableaux). La définition d'un tableau multidimensionnel se réalise de la même manière que celle d'un tableau unidimensionnel si ce n'est que vous devez fournir la taille des différentes dimensions.

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int tabInt[2][3];         // tableau 2 dimensions
  float tabFloat[4][2][3];   // tableau 3 dimensions
  return 0;
}
```

En ce qui concerne l'initialisation nous avons le même comportement que pour les tableaux à une dimension. C'est-à-dire que le tableau n'est pas forcément initialiser à sa création :

```
Breakpoint 1, main (argc=1, argv=0x7fffffffe188) at hello.c:5
5   return 0;
(gdb) print tab
$1 = {{0, 0, 64}, {0, 16, 0}}
```

Il est possible d'initialiser un tableau implicitement ou explicitement, de manière séquentielle ou sélective, et d'omettre la longueur de la première dimension.

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int t1[2][3] = {{1, 2, 3}, {4, 5, 6}};
  int t2[][3] = {{1, 2, 3}, {4, 5, 6}};
  int t3[2][3] = {[0 ... 1] = {1, 2, 3}};
  int t4[][3] = {[0 ... 3] = {1, 2, 3}};
  return 0;
}
```

```
Breakpoint 1, main (argc=1, argv=0x7fffffffe188) at hello.c:9
9   return 0;
(gdb) print t1
$1 = {{1, 2, 3}, {4, 5, 6}}
(gdb) print t2
$2 = {{1, 2, 3}, {4, 5, 6}}
(gdb) print t3
$3 = {{1, 2, 3}, {1, 2, 3}}
(gdb) print t4
$4 = {{1, 2, 3}, {1, 2, 3}, {1, 2, 3}, {1, 2, 3}}
```

Un tableau multidimensionnel est un tableau dont les éléments sont eux-mêmes des tableaux. Vous avez donc besoin d'autant d'indices qu'il y a de dimensions. Illustration.

```c
#include <stdio.h>

int main(int argc, char** argv) {
  int i, j;
  int tab[2][3] = {{1, 2, 3}, {4, 5, 6}};
  printf("[");
  for (i = 0; i < 2; i++) {
    printf("%s[", i ? "," : "");
    for (j = 0; j < 2; j++) printf("%s%d", j ? "," : "", tab[i][j]);
    printf("]");
```

```
  }
  printf("]\n");
  return 0;
}
```

Notons que les données d'un tableau multidimensionnel sont stockées les unes à côté des autres en mémoire : **elles sont rassemblées dans un tableau à une seule dimension**. Considérons le code suivant qui initialise un tableau à 3 dimensions :

```
#include <stdio.h>

int main(int argc, char** argv) {
  int i, j;
  int tab[2][2][3] = {{{1, 2, 3}, {4, 5, 6}}, {{7, 8, 9}, {10, 11, 12}}};
  return 0;
}
```

Si nous utilisons l'utilitaire gdb nous pouvons voir que le tableau est bien représenté ligne par ligne de manière contiguë.

```
Breakpoint 1, main (argc=1, argv=0x7fffffffe188) at hello.c:6
6   return 0;
(gdb) print tab
$1 = {{{1, 2, 3}, {4, 5, 6}}, {{7, 8, 9}, {10, 11, 12}}}
(gdb) print &tab
$2 = (int (*)[2][2][3]) 0x7fffffffe040
(gdb) print *0x7fffffffe040@12
$3 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
```

7.3 Tableaux de littéraux

Il est possible de construire des tableaux « littéraux », c'est-à-dire sans besoin de déclarer une variable. Ces tableaux littéraux ont une classe de stockage automatique, ils sont donc détruits à la fin du bloc qui les a vu naître.

```
#include <stdio.h>

void printInv(int *tab, int sz)
{
  while(--sz >= 0) printf("%d\n",tab[sz]);
}

int main(int argc, char** argv) {
  printInv((int[]) {1,2,3,4,5}, 5);
  return 0;
}
```

Ce sont les même règles que pour l'initialisation qui sont utilisées pour le *cast*.

7.4 Pointeurs et tableaux

Lorsque nous utilisons une variable de type tableau dans une expression, celle-ci est convertie implicitement en un pointeur sur son premier élément. Nous pouvons utiliser une variable de type tableau comme nous l'aurions fait s'il s'agissait d'un pointeur.

```
#include <stdio.h>

int main(void) {
  int tab[] = {1, 2, 3, 4};
  printf("%d\n", *tab);
}
```

Contrairement aux pointeurs, il n'est pas possible d'affecter une valeur à une variable de type tableau, ainsi le code suivant est incorrect :

```c
#include <stdio.h>

int main(void) {
  int t1[3], t2[3];
  t1 = t2;
}
```

Notons que, contrairement aux pointeurs, lorsqu'il est appliqué à une variable de type tableau, l'opérateur & produit comme résultat l'adresse du premier élément du tableau.

```c
#include <stdio.h>

int main(void) {
  int t1[3];
  printf("%p %p\n", &t1, t1);
}
```

Bien que l'adresse référencée par les deux pointeurs soit identique, leurs types sont différents. `t1` est un pointeur sur `int` et `&t1` est un pointeur sur un tableau de 3 `int`.
Comme nous avons pu le voir précédemment, `sizeof` donne la taille totale du tableau (en multiplets) et non la taille d'un pointeur. Cependant, cela n'est pas le cas lorsque nous considérons des pointeurs, même si ces pointeurs pointent sur des tableaux static.

```c
#include <stdio.h>

int main(void) {
  int t[10];
  int *ptr1 = t;
  int(*ptr2)[10] = &t;
  printf("%d %d %d\n", sizeof(t), sizeof(ptr1), sizeof(ptr2)); // affiche : 40 8 8
}
```

Puisqu'un tableau peut être utilisé comme un pointeur sur son premier élément, lorsque vous passez un tableau en argument d'une fonction, celle-ci reçoit un pointeur vers le premier élément du tableau. Cela implique que le contenu du tableau est modifiable à dans la fonction.

```c
#include <stdio.h>

void test(int *tab, int size) {
  int i;
  for (i = 0; i < size; i++) tab[i] = 42;
}

int main(void) {
  int t[] = {[0 ... 9] = 1}, i;
  test(t, 10);
  for (i = 0; i < 10; i++) printf("%d\n", t[i]);
  return 0;
}
```

Un autre point important concerne les tableaux à plusieurs dimensions. Nous avons vu précédemment qu'il était possible de récupérer un élément dans un tableau simplement en enchaînant la notation crochet. Cependant, cela n'est pas le cas si nous considérons la version pointeur du tableau. En effet, le code suivant n'est pas correct :

```c
#include <stdio.h>

int main(void) {
  int t[2][2] = {{1, 2}, {3, 4}}, i, j;
  int **pt = t;
  printf("%d\n", pt[0][0]);
  return 0;
}
```

Chapitre 8

Arithmétique des pointeurs

La valeur d'un pointeur étant en quelque sorte un entier, il est tout à fait possible de modifier un pointeur, c'est-à-dire de changer l'adresse à laquelle il réfère. En fait, il est possible de lui appliquer un certain nombre d'opérateurs arithmétiques classiques :

— Addition
— Soustraction
— Incrémentation/Décrémentation
— Comparaison

8.1 Addition

Lorsqu'une valeur est ajouté à pointeur, la valeur est d'abord multipliée par la taille du type de données, puis ajoutée au pointeur.

```c
#include <stdio.h>

int main(void) {
  int *ptr_i = NULL;
  double *ptr_d = NULL;

  printf("%p %p %p\n", ptr_i, ptr_i + 1, ptr_i + 10);
  printf("%p %p %p\n", ptr_d, ptr_d + 1, ptr_d + 10);
}
```

Ce qui donne comme résultat :

```
(nil) 0x4 0x28
(nil) 0x8 0x50
```

Si nous considérons un tableau static, il est possible d'afficher une case particulière du tableau en utilisant l'addition de la manière suivante :

```c
#include <stdio.h>

int main(void) {
  int t[] = {1, 2, 3, 4, 5, 6};
  int *ptr_i = t;

  printf("%d %d\n", t[2], *(ptr_i + 2));
}
```

L'opération crochet utilisée pour les tableaux peut en fait être vue comme étant équivalent à additionner le pointer avec l'indice et à déréférencer le résultat.

```c
#include <stdio.h>

int main(void) {
  int v = 42;
  unsigned char *ptr = (unsigned char *)&v;
```

```
6
7   printf("%p %p %p %p %p\n", &v, ptr, ptr + 1, ptr + 2, ptr + 3);
8   printf("%d %d %d %d\n", ptr[0], ptr[1], ptr[2], ptr[3]);
9 }
```

Puisque l'addition de deux n'a aucun sens, cette opération n'est pas permise par le compilateur. Par exemple le code suivante ne compilera pas :

```
1 #include <stdio.h>
2
3 int main(void) {
4   int t[] = {1, 2, 3, 4, 5, 6};
5   int *ptr = t;
6   printf("%p\n", ptr + ptr);
7 }
```

8.2 Soustraction

La soustraction avec une valeur entière fonctionne comme pour l'addition avec une valeur entière, c'est-à-dire que lorsqu'une valeur est soustrait à pointeur, la valeur est d'abord multipliée par la taille du type de données, puis soustrait au pointeur.

```
1 #include <stdio.h>
2
3 int main(void) {
4   int t[] = {1, 2, 3, 4, 5, 6};
5   int *ptr_i = &t[2];
6   printf("%p %p %p %d\n", t, ptr_i -2, ptr_i, *(ptr_i - 2));
7 }
```

Comme pour l'addition, il est aussi possible d'utiliser la notation crochet afin de « se déplacer » dans le tableau et de déréférencer l'adresse mémoire associée.

```
1 #include <stdio.h>
2
3 int main(void) {
4   int t[] = {1, 2, 3, 4, 5, 6};
5   int *ptr_i = &t[2];
6   printf("%d %d\n", ptr_i[-2], *(ptr_i - 2));
7 }
```

Nous avons vu que l'addition de deux pointeurs était impossible, mais la soustraction elle est possible et pertinente. Cela permet de calculer le décalage entre les deux adresses.

```
1 #include <stdio.h>
2
3 int main(void) {
4   int t[] = {1, 2, 3, 4, 5, 6};
5   int *ptr_0 = t, *ptr_2 = &t[2];
6   printf("%d\n", ptr_2 - ptr_0);
7 }
```

Pour être plus précis, la soustraction de deux pointeurs n'est possible que lorsqu'ils ont le même type de données. Le résultat est alors la différence entre les adresses des deux pointeurs divisée par le nombre d'octets du type de données.

8.3 Incrémentation/Décrémentation d'un pointeur

L'incrémentation relève de l'addition, c'est-à-dire que lorsqu'un pointeur est incrémenté, il s'incrémente en fait du nombre égal à la taille du type de données.

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

```c
int main(void) {
  int t[] = {1, 2, 3, 4, 5, 6}, i;
  int *ptr = t;
  for (i = 0; i < 6; i++) {
    printf("%p %d\n", ptr, *ptr);
    ptr++;
  }
}
```

Idem pour la décrémentation, elle relève de la soustraction, c'est-à-dire que lorsqu'un pointeur est décrémenté, il se décrémente en fait du nombre égal à la taille du type de données.

```c
#include <stdio.h>

int main(void) {
  int t[] = {1, 2, 3, 4, 5, 6}, i;
  int *ptr = t;
  for (i = 0; i < 6; i++) {
    printf("%p %d\n", ptr, *ptr);
    --ptr;
  }
}
```

8.4 Comparaison de pointeurs

Comme nous l'avons déjà souligné, un pointeur peut être d'une certaine manière assimilé à un entier. Ainsi, nous pouvons comparer deux pointeurs en utilisant les opérateurs de comparaison en C. Le deux programmes précédemment utilisées afin d'illustrer l'incrémentation et la décrémentation peuvent être modifiées de sorte que la déclaration de la variable `i` ne soit pas nécessaire.

```c
#include <stdio.h>

int main(void) {
  int t[] = {1, 2, 3, 4, 5, 6};
  int *ptr = t;
  while (ptr != &t[6]) {
    printf("%p %d\n", ptr, *ptr);
    ptr++;
  }
}
```

```c
#include <stdio.h>

int main(void) {
  int t[] = {1, 2, 3, 4, 5, 6};
  int *ptr = &t[5];
  while (ptr >= t) {
    printf("%p %d\n", ptr, *ptr);
    ptr--;
  }
}
```

Attention : Les deux pointeurs doivent être de même type (sinon le compilateur tousse un peu).

Chapitre 9

Allocation dynamique

Pour le moment la taille des objets que nous avons manipulée était connue à la compilation. Cependant, cela n'est pas toujours le cas en pratique. Considérons la situation où nous souhaitons lire un fichier et stocker le contenu dans un tableau. Une solution pourrait être d'allouer un très gros tableau statique et de sauvegarder le contenu du fichier dans ce tableau. Néanmoins, cette solution n'est pas vraiment satisfaisante pour deux raisons : (1) quelle est la taille du tableau que nous devons allouer et (2) nous faisons quoi de l'espace mémoire que nous n'avons pas utilisé.
En fait, nous ne pouvons pas définir à l'avance combien d'espace mémoire nous aurons besoin, ce qui implique que le compilateur ne peut donc pas faire la réservation de l'espace mémoire automatiquement. C'est donc au programmeur qu'incombe cette réservation de mémoire. Cette opération, appelée allocation de mémoire, doit être faite pendant l'exécution du programme. La différence avec la déclaration de notre tableau précédente, c'est que la taille du fichier et donc la quantité de mémoire à allouer, est variable. L'allocation de la mémoire est donc fait de manière dynamique.

9.1 Allouer de la mémoire

La bibliothèque standard fournit trois fonctions vous permettant d'allouer de la mémoire : `malloc`, `calloc` et `realloc`. Les entêtes pour ces fonctions sont :

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

La fonction `malloc` permet d'allouer une de la mémoire dont la quantité fournie en argument représente un nombre de multiplets. Cette fonction retourne l'adresse du bloc mémoire alloué sous la forme d'un pointeur générique. Nous pouvons par exemple allouer de la mémoire pour un type basique (ici un `int`) de la manière suivante :

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

int main(void) {
  int *ptr = (int *)malloc(sizeof(int));
  *ptr = 42;
  printf("%p %p %d\n", &ptr, ptr, *ptr);
  return 0;
}
```

Il est clair qu'allouer de la mémoire pour un type basique n'a pas vraiment beaucoup de sens, Cela devient beaucoup plus intéressant lorsque nous souhaitons allouer un tableau. Dans l'exemple suivant nous allouons un tableau de 10 entiers :

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

int main(void) {
```

```
  int i = 0;
  int *ptr = (int *)malloc(10 * sizeof(int));

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

Il est important de noter que la fonction `malloc` n'attend pas en paramètre le nombre d'élément du tableau mais la quantité de mémoire nécessaire pour stocker le tableau en mémoire. Autrement dit, pour allouer un tableau de `n` éléments de type `T` il faut demander à `malloc` un bloc mémoire de taille `n * sizeof(T)`.

Notons aussi que la fonction `malloc` n'effectue aucune initialisation. Si vous souhaitez, par exemple, obtenir un tableau où chaque octet est initialisé à zéro vous utiliserez la fonction `calloc`. Cette fonction attend deux arguments : le nombre d'éléments à allouer et la taille de chacun de ces éléments.

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

int main(void) {
  int i = 0;
  int *ptr = (int *)calloc(10, sizeof(int));

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

Dans l'arsenal des fonctions d'allocation, il nous reste encore à étudier la fonction `realloc`. Cette fonction a pour objectif d'étendre un bloc mémoire que vous avez alloué précédemment via un appel à une fonction d'allocation. Pour cela, elle libère un bloc de mémoire précédemment alloué, en réserve un nouveau de la taille demandée et copie le contenu de l'ancien objet dans le nouveau. Deux situations se présente alors à nous :

1. Si la taille demandée est inférieure à celle du bloc d'origine, le contenu de celui-ci sera copié à hauteur de la nouvelle taille. Généralement, si la quantité de mémoire demandée est inférieure à la quantité de mémoire pointée alors le pointeur est retourné directement (mais ce n'est pas une règle absolue).
2. Si la nouvelle taille est supérieure à l'ancienne, l'excédent n'est pas initialisé.

La fonction `realloc` attend deux arguments : l'adresse d'un bloc précédemment alloué à l'aide d'une fonction d'allocation et la taille du nouveau bloc à allouer. L'exemple suivant décrit un cas où l'utilisateur commence par allouer un espace mémoire pour 10 entiers, il l'étend ensuite pour qu'il puisse accueillir 20 entiers, et finalement le réduit pour qu'il ne puisse accueillir que 5 entiers.

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

int main(void) {
  int i = 0;
  int *ptr1 = (int *)calloc(10, sizeof(int));
  printf("%p\n", ptr1);
  int *ptr2 = (int *)realloc(ptr1, 20 * sizeof(int));
  printf("%p\n", ptr2);
  int *ptr3 = (int *)realloc(ptr2, 5 * sizeof(int));
  printf("%p\n", ptr3);
  return 0;
}
```

Notons qu'il est aussi tout à fait possible d'appeler la fonction `realloc` avec comme premier paramètre un pointeur qui pointe sur `NULL`. Dans ce cas, la fonction `realloc` se comporte exactement comme `malloc`.

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

```c
#include <stdlib.h>

int main(void) {
  int i = 0;
  int *ptr = NULL;
  ptr = (int *)realloc(ptr, 10 * sizeof(int));
  for (i = 0; i < 10; i++) ptr[i] = 0;
  for (i = 0; i < 10; i++) printf("%d\n", ptr[i]);
  return 0;
}
```

Dans l'exemple précédent illustrant le cas général de l'utilisation de la fonction `realloc`, nous avons à chaque fois utilisé une nouvelle variable pour stocker le résultat de l'appel de fonction. Il aurait tout à fait été envisageable d'utiliser une seule variable comme dans l'exemple suivant :

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

int main(void) {
  int i = 0;
  int *ptr = (int *)calloc(10, sizeof(int));
  printf("%p\n", ptr);
  ptr = (int *)realloc(ptr, 20 * sizeof(int));
  printf("%p\n", ptr);
  ptr = (int *)realloc(ptr, 5 * sizeof(int));
  printf("%p\n", ptr);
  return 0;
}
```

Néanmoins, cette solution n'est pas totalement satisfaisante dans le cas où l'appel à la fonction `realloc` échoue. Dans ce cas, nous perdons la référence sur l'adresse de la zone mémoire que nous souhaitions réallouer, ce qui peut conduire à ce que nous appelons des fuites de mémoire. Une fuite mémoire est un situation où la mémoire que nous avons alloué n'est plus accessible (plus aucune variable ne pointe sur la zone mémoire). Dans ce cas, il n'y a pas de mécanisme en C permettant de récupérer la mémoire alloué et donc cette mémoire est perdue. Pour observer si vous avez des fuites mémoire vous pouvez utiliser l'utilitaire `valgring`. Considérons le code suivant :

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

int main(void) {
  int *ptr = (int *)calloc(10, sizeof(int));
  float *f = (float *)malloc(12 * sizeof(float));
  return 0;
}
```

Si nous compilons et exécutons `valgrind` nous obtenons les informations suivantes :

```
$ gcc -o exec test.c
$ valgrind ./exec
==30144== Memcheck, a memory error detector
==30144== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==30144== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==30144== Command: ./exec
==30144==
==30144==
==30144== HEAP SUMMARY:
==30144==     in use at exit: 88 bytes in 2 blocks
==30144==   total heap usage: 2 allocs, 0 frees, 88 bytes allocated
==30144==
==30144== LEAK SUMMARY:
==30144==    definitely lost: 88 bytes in 2 blocks
==30144==    indirectly lost: 0 bytes in 0 blocks
==30144==      possibly lost: 0 bytes in 0 blocks
==30144==    still reachable: 0 bytes in 0 blocks
```

```
==30144==          suppressed: 0 bytes in 0 blocks
==30144== Rerun with --leak-check=full to see details of leaked memory
==30144==
==30144== For lists of detected and suppressed errors, rerun with: -s
==30144== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Il est même possible d'obtenir les lignes où ces allocations ont été réalisé :

```
$ gcc -g -o exec test.c
$ valgrind --leak-check=full ./exec
==30256== Memcheck, a memory error detector
==30256== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==30256== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==30256== Command: ./exec
==30256==
==30256==
==30256== HEAP SUMMARY:
==30256==     in use at exit: 88 bytes in 2 blocks
==30256==   total heap usage: 2 allocs, 0 frees, 88 bytes allocated
==30256==
==30256== 40 bytes in 1 blocks are definitely lost in loss record 1 of 2
==30256==    at 0x484DA83: calloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==30256==    by 0x109183: main (test.c:6)
==30256==
==30256== 48 bytes in 1 blocks are definitely lost in loss record 2 of 2
==30256==    at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==30256==    by 0x109191: main (test.c:7)
==30256==
==30256== LEAK SUMMARY:
==30256==    definitely lost: 88 bytes in 2 blocks
==30256==    indirectly lost: 0 bytes in 0 blocks
==30256==      possibly lost: 0 bytes in 0 blocks
==30256==    still reachable: 0 bytes in 0 blocks
==30256==         suppressed: 0 bytes in 0 blocks
==30256==
==30256== For lists of detected and suppressed errors, rerun with: -s
==30256== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Afin d'éviter ce genre de situations nous verrons dans la section suivante que nous pouvons libérer une zone mémoire allouée dynamiquement. À partir de maintenant dès que vous allez allouer de la mémoire, vous devrez penser en même temps au moment où vous allez la libérer.

9.2 Libérer de la mémoire

La fonction `free` permet de libérer un bloc précédemment alloué par une fonction d'allocation dont l'adresse est fournie en argument.

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

int main(void) {
  int *ptr = (int *)calloc(10, sizeof(int));
  float *f = (float *)malloc(12 * sizeof(float));
  free(ptr);
  free(f);
  return 0;
}
```

Dans le cas où un pointeur `NULL` lui est fourni, elle n'effectue aucune opération. Regardons à présent la sortie de l'utilitaire `valgrind` sur le code précédent :

```
$ gcc -g -o exec test.c
$ valgrind --leak-check=full ./exec
==33510== Memcheck, a memory error detector
==33510== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
```

```
==33510== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==33510== Command: ./exec
==33510==
==33510==
==33510== HEAP SUMMARY:
==33510==     in use at exit: 0 bytes in 0 blocks
==33510==   total heap usage: 2 allocs, 2 frees, 88 bytes allocated
==33510==
==33510== All heap blocks were freed -- no leaks are possible
==33510==
==33510== For lists of detected and suppressed errors, rerun with: -s
==33510== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

9.3 Les tableaux multidimensionnels

Il existe deux méthodes pour allouer un tableau multidimensionnel :

— l'allocation d'un seul bloc de mémoire (comme pour les tableaux statiques) ;
— l'allocation de plusieurs tableaux eux-mêmes référencés par les éléments d'un autre tableau.

9.3.1 Allocation d'un bloc

Comme pour les tableaux statiques, nous pouvons allouer un bloc mémoire contiguë permettant de représenter en mémoire notre tableau multidimensionnel. Néanmoins, contrairement au cas des tableaux multidimensionnels statiques, il n'est pas possible d'accéder à un élément du tableau avec un enchaînement de crochets. En fait, comme le montre l'exemple suivant, il est nécessaire de réaliser une partie des calculs d'adresse.

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

int main(void) {
  int i, j, cpt = 0;
  int nbL = 10, nbC = 20;
  int *tab = (int *)malloc(nbC * nbL * sizeof(int));
  for (i = 0; i < nbL; i++)
    for (j = 0; j < nbC; j++)
      tab[i * nbC + j] = cpt++;
  free(tab);
  return 0;
}
```

Comme nous pouvons le voir, le calcul de l'adresse est effectué en multipliant le premier indice par la longueur de la première dimension, ce qui permet de sélectionner la bonne ligne. Ensuite, il ne reste plus qu'à sélectionner le bon élément de la colonne à l'aide du second indice. Cette solution à l'avantage d'allouer un bloc mémoire contiguë, ce qui de manière générale permet d'obtenir un gain en performance lors de l'exécution d'un algorithme nécessitant de nombreux accès à ce type de tableau.

9.3.2 Allocation de plusieurs tableaux

Une autre solution est d'allouer un tableau de pointeurs, qui contiendra pour chaque case un tableau de la dimension inférieure alloué dynamiquement, et cela tant que la dimension est supérieur à un. Dans le cas d'un tableau à deux dimensions, cela signifie allouer un tableau de pointeurs où chaque élément reçoit l'adresse d'un tableau également alloué dynamiquement. Cette technique nous permet d'accéder aux éléments des différents tableaux de la même manière que pour un tableau multidimensionnel statique. En effet, ici nous pouvons directement accéder à la dimension avec la notation crochet. L'exemple précédent peut être réécrit de la manière suivante :

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

```
3
4  int main(void) {
5    int i, j, cpt = 0;
6    int nbL = 10, nbC = 20;
7    int **tab = (int **)malloc(nbL * sizeof(int *));
8    for (i = 0; i < nbL; i++) {
9      tab[i] = (int *) malloc(nbC * sizeof(int));
10     for (j = 0; j < nbC; j++)
11       tab[i][j] = cpt++;
12   }
13
14   for (i = 0; i < nbL; i++) free(tab[i]);
15   free(tab);
16   return 0;
17 }
```

Ici libérer la mémoire est un peu plus complexe puisque cela nécessite de parcourir le tableau principal pour d'abord libérer la mémoire de ses éléments. Une fois que la mémoire que chaque sous tableau a été libéré, alors il est possible de libérer la mémoire du tableau principale (et pas l'inverse).

9.4 Les tableaux de taille variable

En préambule nous avons souligné qu'il était impossible d'allouer de la mémoire de manière dynamique autrement que via l'appel à une des fonctions que nous avons énoncé précédemment. En fait, cette assertion n'est pas totalement correcte. Il est tout à fait possible d'allouer de la mémoire dynamiquement avec la notation crochet. Plus précisément il est possible de considérer des tableaux de longueur variable en employant une variable à la place d'une constante entière.

```
1  #include <stdio.h>
2
3  int main(void) {
4    int i, j, cpt = 0;
5    int nbL = 10, nbC = 20;
6    int tab[nbL][nbC];
7    for (i = 0; i < nbL; i++)
8      for (j = 0; j < nbC; j++) tab[i][j] = cpt++;
9    return 0;
10 }
```

Ainsi, si la taille du tableau ne peut pas être déterminée à la compilation, le compilateur va en fait ajouter des instructions afin d'allouer et de libérer de la mémoire dynamiquement. Notons qu'il est donc impossible d'initialiser le tableau lors de sa définition. Cela tient au fait que le compilateur ne peut effectuer aucune vérification quant à la taille de la liste d'initialisation puisque nous ne connaissons pas avant l'exécution la valeur de la variable qui servira à définir la longueur du tableau.
En ce qui concerne l'utilisation des tableaux à longueur variable, c'est presque tout comme les tableaux classiques. Néanmoins, il existe tout de même quelques subtilités. Ainsi, si nous souhaitons utiliser un tableau à longueur variable dans la déclaration d'une fonction, il est nécessaire que le compilateur sache retrouver la taille des différentes dimensions. Pour cela, il suffit de donner les variables définissant les dimensions avant de spécifier le tableau. Un exemple sera plus parlant :

```
1  #include <stdio.h>
2
3  void print(int l, int c, int tab[l][c]) {
4    int i, j;
5    for (i = 0; i < l; i++) {
6      for (j = 0; j < c; j++) printf("%d ", tab[i][j]);
7      printf("\n");
8    }
9  }
10
11 int main(void) {
```

```
12   int i, j, cpt = 0;
13   int nbL = 10, nbC = 20;
14   int tab[nbL][nbC];
15   for (i = 0; i < nbL; i++)
16     for (j = 0; j < nbC; j++) tab[i][j] = cpt++;
17   print(nbL, nbC, tab);
18   return 0;
19 }
```

Après avoir vu comment il était facile de laisser le soin au compilateur d'allouer dynamiquement (et surtout désallouer la mémoire allouée), nous sommes en droit de nous poser la question : À quoi cela sert-il d'utiliser les fonctions d'allocation de mémoire que nous avons vu dans la première partie de ce chapitre ? En fait, les tableaux à longueur variable souffrent de quelques limitations.
La première limitation est que la classe de stockage des tableaux de longueur variable est automatique. En effet, puisque le compilateur libérer la mémoire allouée, il est donc nécessaire de le moment où la mémoire sera libérée. Ainsi, comme pour toute variable automatique, il n'y a pas de sens de retourner l'adresse d'un tableau de longueur variable puisque celui-ci n'existera plus une fois la fonction terminée. Par exemple le code suivant est incorrect :

```
1  #include <stdio.h>
2
3  int *getTableau(int n) {
4    int tab[n];
5    return tab;
6  }
7
8  int main(void) {
9    int *tab = getTableau(10);
10   return 0;
11 }
```

Une autre limitation tient au fait que la taille d'un tableau que nous pouvons allouer sur la pile est très largement inférieure à celle que nous pouvons allouer sur le tas. Cela peut être très limitant pour certaines applications !
Un point que nous avons passé sous silence jusqu'il est la gestion des erreurs, nous verrons cela au chapitre suivant ne vous inquiétez pas ...En fait, lorsque la fonction `malloc` échoue à allouer un morceau de mémoire alors elle retournera `NULL` et il sera alors possible de réagir en conséquence. Dans le cas des tableaux à longueur variable, il est impossible de savoir si l'allocation c'est bien passée et donc de réagir en conséquence (cela peut être une grosse faille de sécurité).

Chapitre 10

Gestion des erreurs

Jusqu'ici nous ne nous sommes pas attardé sur la question de ce qui se passe si les fonctions que nous avons appelé échouées. Nous verrons que dans ce genre de situation, le langage C offre plusieurs mécanisme afin d'alerter et de réagir de manière adéquate.

10.1 Détection d'erreurs

Il est clair que pour gérer d'éventuelles erreurs lors de l'exécution, il faut avant tout les détecter. Cela implique qu'une fonction susceptible de produire une erreur lors de l'exécution doit être en mesure de vous avertir d'une manière ou d'une autre. En fait, il existe deux manières de le faire :

1. annoncer l'erreur via le retour de la fonction ;
2. annoncer l'erreur en modifiant une variable global.

La première solution est celle est qui est utilisé par la plupart des fonctions. Par exemple lorsque la fonction `malloc` échoue elle retourne la valeur `NULL`. Ainsi, en vérifiant la valeur du pointeur retournée il est possible de vérifier si l'allocation de mémoire c'est bien déroulée.

```c
#include <stdio.h>

int main(void) {
  int *tab = (int *)malloc(10 * sizeof(int));
  if (!tab) return 1;
  free(tab);
  return 0;
}
```

Lorsque nous observons le code précédent, nous remarquons que la fonction `main` suit aussi ce schéma. En fait, deux constantes sont définies dans l'en-tête `stdlib.h` permettant de parer aux deux éventualités :

— `EXIT_SUCCESS` qui indique que tout s'est bien passé ;
— `EXIT_FAILURE` qui indique un échec du programme.

Il n'est pas toujours possible d'utiliser le retour d'une fonction afin d'annoncer qu'une erreur est survenue. Dans ce cas, il est nécessaire d'utiliser un autre canal de communication afin de signaler qu'une erreur est apparue. La bibliothèque standard fourni une variable globale nommée `errno`, déclarée dans l'en-tête `errno.h>`, qui permet à différentes fonctions d'indiquer une erreur en modifiant la valeur de celle-ci. Dans ce cas la valeur zéro indique qu'aucune erreur est survenue. Considérons l'exemple suivante qui appel la fonction `sqrt` avec une valeur négative.

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

int main(void) {
  errno = 0;
  double v = sqrt(-1);
  printf("%lf %d\n", v, errno);
```

```
9
10  return 0;
11 }
```

Il est important de remettre à zéro la valeur de la variable `errno` lorsque nous souhaitons l'utiliser ensuite, ceci afin de vous assurer qu'elle ne contient pas la valeur qu'une autre fonction lui a assignée auparavant.

10.2 Annoncer une erreur

Pour annoncer à l'utilisateur il est toujours possible de l'annoncer sur la sortie standard. Cependant, cela n'est pas toujours le meilleur canal de communication puisque certaines erreurs ne sont pas critiques, et hurler dans le même canal que le canal de communication standard peut être assez fatiguant pour l'utilisateur. En fait, lorsque votre programme est exécuté trois canaux de communications sont initialisés : `stdin`, `stdout` et `stderr`. Le canal de communication `stdin` est celui utilisé pour récupérer des informations de la part de l'utilisateur (cela peut être un humain ou un autre programme). En ce qui concerne `stdout` et `stderr`, ils servent tous le deux à afficher un message. La différence entre les deux canaux est que `stdout` affiche sur la sortie standard, tandis que `stderr` affiche sur la sortie standard des erreurs (nous verrons tous cela plus en détail dans le chapitre 12).

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5    int *tab = (int *)malloc(10 * sizeof(int));
6    if (!tab) {
7      fprintf(stderr, "Problème lors de l'appel à malloc\n");
8      return EXIT_FAILURE;
9    }
10   free(tab);
11   return EXIT_SUCCESS;
12 }
```

Bien que cette manière d'annoncer une erreur soit acceptable dans de nombreuses situations, elle n'aide pas à comprendre pourquoi cette erreur a eu lieu. Comme nous le verrons dans la section suivante, pour les fonctions de la librairie standard il est possible d'obtenir plus d'informations.

10.2.1 Les fonctions `strerror` et `perror`

La fonction `strerror`, déclarée dans l'en-tête `string.h>` prend en paramètre un entier correspondant à l'erreur qui est survenue (en fait nous passerons systématiquement la variable `errno`) et retourne une chaîne de caractères correspondant à la valeur entière fournie en argument. Il est ainsi possible d'obtenir plus de détails quand une fonction standard rencontre un problème.

```
1  #include <errno.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  int main(void) {
7    int *tab = (int *)malloc(10 * sizeof(int));
8    if (!tab) {
9      fprintf(stderr, "Problème lors de l'appel à malloc : %s\n", strerror(errno));
10     return EXIT_FAILURE;
11   }
12   free(tab);
13   return EXIT_SUCCESS;
14 }
```

Grâce à la fonction `perror`, déclarée dans l'en-tête `stdio.h`, il est possible de réaliser le même traitement de manière plus concise. Cette fonction écrit sur le flux d'erreur standard la chaîne de caractères fournie

en argument, suivie du caractère : et du retour de la fonction `strerror` avec comme argument la valeur de la variable `errno`.

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

int main(void) {
  int *tab = (int *)malloc(10 * sizeof(int));
  if (!tab) {
    perror("Problème lors de l'appel à malloc\n");
    return EXIT_FAILURE;
  }
  free(tab);
  return EXIT_SUCCESS;
}
```

10.2.2 Terminaison d'un programme

Dans certaines situations une erreur peut être critique et donc conduire à l'arrêt du programme. Pour stopper un programme de manière « brutale » il est possible de faire appel à la fonction `exit` qui est déclarée dans l'en-tête `stdlib.h`. L'appel de cette fonction revient en quelque sorte à quitter la fonction `main` à l'aide de l'instruction `return`. Cette fonction attend un argument un entier identique à celui fournie comme opérande de l'instruction `return` (et donc avec le même but). Ainsi, il est possible de mettre fin directement à l'exécution de votre programme de manière abrupte.

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

void principal() {
  int *tab = (int *)malloc(10 * sizeof(int));
  if (!tab) {
    fprintf(stderr, "Problème lors de l'appel à malloc\n");
    exit(EXIT_FAILURE);
  }
  free(tab);
}

int main(void) {
  principal();
  return EXIT_SUCCESS;
}
```

Il est aussi possible d'être plus précis et d'appeler la fonction `abort`, déclarée dans l'en-tête `stdlib.h`, pour spécifier qu'une condition non prévue par le programmeur est survenue. Cette situation est à distinguer d'une terminaison normale survenue suite à une erreur prévue (communication avec l'utilisateur par exemple). La terminaison anormale est généralement utilisée lors de la phase de développement d'un logiciel afin de faciliter la détection d'erreurs de programmation. En effet, elle permet souvent de produire d'une image mémoire (i.e. un fichier contenant l'état des registres et de la mémoire d'un programme) qui pourra être analysée à l'aide d'un débogueur.

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

void principal() {
  int *tab = (int *)malloc(10 * sizeof(int));
  if (!tab) {
    fprintf(stderr, "Problème lors de l'appel à malloc\n");
    abort();
  }
  free(tab);
}

int main(void) {
```

```
14   principal();
15   return EXIT_SUCCESS;
16 }
```

10.3 Les assertions

La macrofonction `assert`, définie dans l'en-tête `assert.h`, est utilisée pour placer des tests à certains points d'un programme. Si un de ces tests s'avère faux, alors un message d'erreur est affiché et la fonction `abort` est appelée afin de produire une image mémoire. Cette manière de penser un programme est très utile pour détecter rapidement des erreurs au sein d'un programme lors de sa phase de développement.

```
1 #include <assert.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(int argc, char* argv[]) {
6   assert(argc == 2);
7   printf("le paramètre est : %s\n", argv[1]);
8   return EXIT_SUCCESS;
9 }
```

Ce n'est pas parce que votre programme n'est plus en mode développement qu'il faut supprimer vos assertions. En fait, conserver vos `assert` vous permettra d'être « sûr » que votre programme à le comportement que vous attendez. Néanmoins, la réalisation de ces tests supplémentaires à un coût que vous ne voulez peut être pas payer lorsque votre programme est en production. Puisque, `assert` est une macrofonction il est possible il est possible de « supprimer » ces assertions sans modifier votre code en ajoutant simplement l'option `-DNDEBUG`lors de la compilation (nous verrons comment cela fonctionne en détail lorsque nous étudierons les introductions du préprocesseur au chapitre 13).

```
1 $ gcc -o exec test.c -DNDEBUG
```

10.4 Gérer les ressources en cas d'erreurs

Les ressources d'un ordinateur sont limitées, il faut donc libérer ces dernières et cela même lorsqu'une erreur survient. Considérons la situation suivante :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char *argv[]) {
5   int i, j, l = 10, c = 10;
6   int **tab = (int **)malloc(l * sizeof(int *));
7   if (!tab) {
8     perror("Problème lors de l'appel à malloc\n");
9     return EXIT_FAILURE;
10  }
11
12  for (i = 0; i < l; i++) {
13    tab[i] = (int *)malloc(c * sizeof(int));
14    if (!tab[i]) {
15      perror("Problème lors de l'appel à malloc\n");
16      return EXIT_FAILURE;
17    }
18  }
19
20  for (i = 0; i < l; i++)
21    for (j = 0; j < c; j++) tab[i][j] = i * c + j;
22
23  for (i = 0; i < l; i++) free(tab[i]);
24  free(tab);
25
```

```
26   return EXIT_SUCCESS;
27 }
```

Dans le code précédant, lorsque nous récupérons le retour de la fonction `malloc` et lorsque celui-ci annonce qu'il y a eu une erreur alors nous quittons le programme (nous affichons même un petit message de circonstance). Bien que cela soit le comportement recherché, il y a un soucis lié au fait que nous avons alloué des ressources que nous n'avons pas libérées. Il faut donc trouver un mécanisme permettant de résoudre ce problème de manière « élégante ». Pour cela nous pouvons utiliser l'instruction `goto`. Le programme précédent peut donc être réécrit comme :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[]) {
5    int i, j, l = 10, c = 10;
6    int **tab = (int **)malloc(l * sizeof(int *));
7    if (!tab) {
8      perror("Problème lors de l'appel à malloc\n");
9      goto delivrer;
10   }
11
12   for (i = 0; i < l; i++) {
13     tab[i] = (int *)malloc(c * sizeof(int));
14     if (!tab[i]) {
15       perror("Problème lors de l'appel à malloc\n");
16       goto liberer;
17     }
18   }
19
20 liberer:
21   for (i = 0; i < l; i++)
22     for (j = 0; j < c; j++) tab[i][j] = i * c + j;
23
24   for (i = 0; i < l; i++) free(tab[i]);
25   free(tab);
26
27 delivrer:
28   return EXIT_SUCCESS;
29 }
```

Chaque situation est singulière, vous devrez donc vous adapter!

Chapitre 11

Les types composés

Au chapitre 2 nous avons vu l'ensemble des types disponibles en C, les autres types que vous utiliserez seront en fait de types composés, qui permettent de représenter des ensembles de données organisées.

11.1 Les énumérations

Avant de parler de structure nous allons nous attarder sur la définition et l'utilisation d'énumération. Une énumération se définit à l'aide du mot clef `enum` de la manière suivante :

```
enum couleur {VERT, JAUNE, ROUGE};
```

Avec cette déclaration un nouveau type est créé, ici le type couleur, et un ensemble de constantes, ici les constantes `VERT`, `JAUNE` et `ROUGE`. Les constantes sont en fait associées à des entiers (`int` pour les intimes). Une fois déclaré, il est possible d'utiliser ce type et les constantes directement dans le code.

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

enum couleur { VERT, JAUNE, ROUGE};

int main(int argc, char *argv[]) {
  enum couleur c = VERT;
  printf("%d %d %d\n", c, JAUNE, ROUGE); // affiche 0 1 2
  return EXIT_SUCCESS;
}
```

Par défaut les valeurs des constantes sont calculées, et la valeur d'une constante est attribuée en fonction de la valeur de la constante précédente en ajoutant un. Pour la première constante sa valeur est dans ce cas la valeur zéro. Il est aussi possible de spécifier une valeur à certaines ou à toutes les constantes. Pour spécifier, une valeur il suffit d'utiliser l'opérateur d'affectation dans la définition.

```
enum couleur {VERT = 42, JAUNE = 51, ROUGE = 1664};
```

Dans le cas où certaines constantes sont spécifiées et d'autres non, la valeur des constantes non spécifiées sont toujours calculées en considérant la valeur des constantes voisines.

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

enum couleur { VERT, JAUNE = 42, ROUGE};

int main(int argc, char *argv[]) {
  enum couleur c = VERT;
  printf("%d %d %d\n", c, JAUNE, ROUGE); // affiche 0 42 43
  return EXIT_SUCCESS;
}
```

Il est aussi possible de créer un `enum` sans spécifier de nom. Dans ce cas, uniquement les constantes énumérées sont produites.

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

enum { VERT, JAUNE, ROUGE};

int main(int argc, char *argv[]) {
  printf("%d %d %d\n", VERT, JAUNE, ROUGE); // affiche 0 1 2
  return EXIT_SUCCESS;
}
```

11.2 Les structures

Une structure (ou un enregistrement) est un regroupements d'objets. Pour définir une structure nous utilisons le mot clef `struct` de la manière suivante :

```c
struct nom_de_la_structure
{
  /* Liste des objets composants la structure */
};
```

Les composantes d'une structure, appelées champs ou membres, peuvent être de différents types, même des types que nous avons construit précédemment. Chaque composante de la structure se voit associée un nom, et les règles à respecter pour ces dernières sont les mêmes que pour les noms de variable et de fonctions. En ce qui concerne le nom de la structure, aussi appelée étiquette, ce sont encore les même règles de nommage qui sont utilisées. Voici un exemple d'une structure permettant de représenter un point :

```c
struct point
{
   int x;
   int y;
};
```

Comme pour la déclaration de variables, il est possible de définir plusieurs champs sur une même ligne :

```c
struct point
{
   int x, y;
};
```

Une fois que nous avons déclaré une structure, il est possible de l'utiliser comme n'importe quelle autre variable. Voici un exemple où nous déclarons une nouvelle structure, appelée `triangle`, composée de trois points et où nous déclarons une variable de type `triangle` dans la fonction `main`.

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

struct point {
  int x, y;
};

struct triangle
{
   struct point p1, p2, p3;
};


int main(int argc, char *argv[]) {
  struct triangle t;
  return EXIT_SUCCESS;
}
```

Puisque les champs d'une structure sont spécifiés à la compilation, la taille de la structure est donc fixe et il est alors tout à fait possible de déclarer des tableaux de structures de la manière suivante :

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

struct point {
  int x, y;
};

struct triangle
{
   struct point p[3];
};


int main(int argc, char *argv[]) {
  struct triangle t;
  return EXIT_SUCCESS;
}
```

En ce qui concerne l'initialisation d'une structure, il est possible de réaliser cela de trois manières :

— initialisation séquentielle ;
— initialisation sélective ;
— initialisation via l'accès aux membres de la structure.

L'initialisation séquentielle permet de spécifier une valeur pour un ou plusieurs membres de la structure en suivant l'ordre de la définition. L'exemple suivant illustre notre propos en initialisant un point :

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

struct point {
  int x, y;
};

int main(int argc, char *argv[]) {
  struct point p = {0.5, 4};
  return EXIT_SUCCESS;
}
```

En pratique, nous ne souhaitons pas toujours initialiser tous les champs de manière séquentielle (surtout qu'il peut y en avoir beaucoup). Il est possible de recourir à une initialisation sélective en spécifiant explicitement le ou les champs à initialiser. Par exemple, il est possible d'assigner uniquement une valeur au champs `x` de la structure `point` de la manière suivante :

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

struct point {
  int x, y;
};

int main(int argc, char *argv[]) {
  struct point p = {.x = 0.5};
  return EXIT_SUCCESS;
}
```

Il aussi possible de mixer les deux dernières initialisations. Dans ce cas, l'initialisation séquentielle reprend au dernier membre désigné par une initialisation sélective. Voici un exemple avec une structure permettant de représenter l'heure :

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

struct temps {
  int heures, minutes, secondes;
};

```

```c
int main(int argc, char *argv[]) {
  struct temps t = {1, .minutes = 24, 14};
  return EXIT_SUCCESS;
}
```

Comme pour les tableaux il est possible d'utiliser des structures littérales. Ainsi il est possible d'affecter une valeur à une structure en dehors de son initialisation en utilisant la notation avec des accolades. Néanmoins, il vous sera nécessaire de réaliser un `cast` pour spécifier que la structure littérale que vous êtes en train de construire est bien du type de la variable que vous souhaitez affecter. Voici un exemple :

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

struct point {
  int x, y;
};

int main(int argc, char *argv[]) {
  struct point p = {0,0};
  p = (struct point) {1,1};
  return EXIT_SUCCESS;
}
```

Attention : l'utilisation du `cast` est obligatoire pour que le compilateur soit en mesure de connaître la taille des éléments de la structure littérale qu'il doit construire. Par exemple, le code suivant ne compilera pas :

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

struct point {
  int x, y;
};

int main(int argc, char *argv[]) {
  struct point p = {0,0};
  p = {1,1};
  return EXIT_SUCCESS;
}
```

Il est à présent temps d'accéder aux champs de notre structure (pour l'utiliser cela peut être utile). Pour cela nous utilisons l'opérateur `.` suivi du nom du champs visé. Cette syntaxe peut être utilisée aussi bien pour obtenir la valeur d'un champ que pour en modifier le contenu. Ainsi l'initialisation d'une structure peut se faire de la manière suivante :

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

struct point {
  int x, y;
};

int main(int argc, char *argv[]) {
  struct point p;
  p.x = 0.5;
  p.y = 4;
  return EXIT_SUCCESS;
}
```

Comme pour n'importe quel type de variable, il est possible de déclarer un pointeur sur une structure. Dans ce cas, l'accès aux membres peut être réalisé de deux manières :

- soit en appliquant l'opérateur `*` pour récupérer la structure et ensuite en sélectionnant le champs avec l'opérateur `.` ;
- il est aussi possible de réaliser les deux opérations précédentes en une seule opération en utilisant l'opérateur `->`.

L'exemple suivant illustre les deux méthodes d'accès aux membres lorsque nous avons un pointeur sur une structure `point` :

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

struct point {
  int x, y;
};

int main(int argc, char *argv[]) {
  struct point p = {1, 2};
  struct point *ptr_p = &p;
  printf("%d %d\n", (*ptr_p).x, ptr_p->y);
  return EXIT_SUCCESS;
}
```

Notons que, étant donné que l'opérateur `.` s'applique prioritairement à l'opérateur `*`, l'utilisation des parenthèses n'est pas une option. En effet, le code suivant ne compilera pas :

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

struct point {
  int x, y;
};

int main(int argc, char *argv[]) {
  struct point p = {1, 2};
  struct point *ptr_p = &p;
  printf("%d %d\n", *ptr_p.x, ptr_p->y);
  return EXIT_SUCCESS;
}
```

Notons aussi que l'opérateur d'adressage peut s'appliquer aussi bien à une structure qu'à un de ses membres. Ainsi, il est possible de passer l'adresse de champs d'une structure à une fonction pour les initialiser :

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

struct point {
  int x, y;
};

void init(int *v) { *v = 4; }

int main(int argc, char *argv[]) {
  struct point p = {1, 2};
  init(&p.x);
  printf("%d %d\n", p.x, p.y); // affiche 4 2
  return EXIT_SUCCESS;
}
```

En ce qui concerne le passage d'une structure en paramètre d'une fonction, il est important de se rappeler qu'en C les variables sont passées par valeur, cela implique que chacun des membres est copié. Cela peut causer des pertes de performances si vous manipulez de grosses structures. Dans ce contexte, le passage par adresse d'une structure est particulièrement propice.

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

struct point {
  int x, y;
};

```

```c
void init(struct point *ptr) { *ptr = (struct point){0, 0}; }

int main(int argc, char *argv[]) {
  struct point p = {1, 2};
  init(&p);
  printf("%d %d\n", p.x, p.y);  // affiche 0 0
  return EXIT_SUCCESS;
}
```

Jusqu'à présent nous avons toujours déclaré nos structures en dehors de toutes fonctions, c'est-à-dire de manière globale. Cependant, il est possible de définir une structure à l'intérieur d'un bloc et ainsi de limiter sa portée, comme pour les définitions de variables. L'exemple suivant illustre notre propos :

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

struct point {
  int x, y;
};

void testPoint() {
  struct point {
    int x, y, z;
  };

  struct point p = {1, 2, 3};
  printf("%d %d %d\n", p.x, p.y, p.z);  // affiche 1 2 3
}

int main(int argc, char *argv[]) {
  struct point p = {1, 2};
  testPoint();
  printf("%d %d\n", p.x, p.y);  // affiche 1 2
  return EXIT_SUCCESS;
}
```

Puisqu'une structure est en fait un nouveau type, celle-ci peut être placée là où est attendu un type dans une définition de variable. Nous pouvons donc modifier l'exemple précédent en déclarant directement la variable `p` de la fonction `testPoint` en même temps que nous spécifions la structure `point`.

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

struct point {
  int x, y;
};

void testPoint() {
  struct point {
    int x, y, z;
  } p = {1, 2, 3};
  printf("%d %d %d\n", p.x, p.y, p.z);  // affiche 1 2 3
}

int main(int argc, char *argv[]) {
  struct point p = {1, 2};
  testPoint();
  printf("%d %d\n", p.x, p.y);  // affiche 1 2
  return EXIT_SUCCESS;
}
```

Comme pour les fonctions doublement récursives, il peut exister des situations où deux structures sont interdépendantes. Dans cette situation, la déclaration de la première structure (première en terme de déclaration) a besoin de savoir que la seconde structure existe. Pour réaliser cela il est possible de déclarer une de structure sans le corps de cette dernière (en fait vous passez un contrat avec le compilateur où vous

stipulez que cette structure existera à la fin). Cependant, puisqu'une déclaration de structure crée un type dit incomplet, il ne peut pas être utilisé pour définir une variable (puisque les membres qui composent la structure sont inconnus). Par exemple le code suivant ne compilera pas :

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

struct structureB;

struct structureA {
  struct structureB b;
};

struct structureB {
  struct structureA a;
};

int main(int argc, char *argv[]) { return EXIT_SUCCESS; }
```

En fait, ceci n'est utilisable que pour définir des pointeurs. Ainsi, en modifiant l'exemple précédent en considérant des pointeurs à la place des structures nous obtenons le résultat attendu :

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

struct structureB;

struct structureA {
  struct structureB *b;
};

struct structureB {
  struct structureA *a;
};

int main(int argc, char *argv[]) { return EXIT_SUCCESS; }
```

En fait, c'est le même mécanisme que vous devrez mettre en place lorsque vous voudrez définir une structure récursive. Prenons comme exemple la déclaration d'une liste chaînée d'entiers :

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

struct list;

struct list {
  int val;
  struct list *next;
};

int main(int argc, char *argv[]) { return EXIT_SUCCESS; }
```

Avant de conclure cette section et d'étudier les champs de bits, attardons nous quelques instants sur la représentation en mémoire de nos structures. Connaître la quantité de mémoire que représente une structure est une condition essentielle pour allouer de la mémoire (via `malloc` et consorts) pour la stocker. Pour réaliser cela il suffit d'utiliser l'opérateur `sizeof`. Considérons un exemple :

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

struct point {
  int x, y;
};

int main(int argc, char *argv[]) {
  printf("taille = %d\n", sizeof(struct point));
```

```
  return EXIT_SUCCESS;
}
```

Le programme précédent affichera 8, ce qui semble logique puisque notre structure comporte deux entiers de 4 octets chacun. Pouvons nous alors déduire la taille d'une structure en sommant la taille des champs qui la compose? La réponse à cette question est non. Pour illustrer cela considérons un autre exemple :

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

struct etudiant {
  int id;
  char genre;
  int age;
};

int main(int argc, char *argv[]) {
  printf("taille = %d\n", sizeof(struct etudiant));
  return EXIT_SUCCESS;
}
```

Contrairement à ce que pourrait suggérer notre premier exemple, le programme précédent n'affichera pas 9 mais 12. La raison est que le compilateur va ajouter des octets de bourrage afin d'aligner la mémoire sur la taille d'un mot (la taille d'un mot dépend de l'architecture, généralement 32-bits ou 64-bits). Pour être plus précis, il est uniquement possible de lire ou écrire des blocs de données de la taille d'un mots mémoire. Comme corollaire, nous avons qu'il est plus efficace que les données à lire ou à écrire soient contenues dans un de ces mots mémoires. Afin de s'en persuader, supposons que le compilateur ne fasse pas de bourrage. Dans ce cas, pour lire l'âge de l'étudiant nous aurons besoin de deux cycles CPU au lieu d'un seul. Nous pouvons aussi utiliser l'utilitaire `gdb` afin d'afficher les adresses des différents champs :

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

struct etudiant {
  float taille;
  char genre;
  int age;
};

int main(int argc, char *argv[]) {
  struct etudiant et1;
  printf("taille = %d\n", sizeof(struct etudiant));
  return EXIT_SUCCESS;
}
```

```
$ gcc -g -o exec hello.c
$ gdb ./exec
(gdb) break 13
(gdb) run
(gdb) print &et1
$1 = (struct etudiant *) 0x7fffffffe064
(gdb) print &et1.taille
$2 = (float *) 0x7fffffffe064
(gdb) print &et1.genre
$3 = 0x7fffffffe068 ""
(gdb) print &et1.age
$4 = (int *) 0x7fffffffe06c
```

Le octets de bourrage ne sont pas ajouter n'importe comment, en fait il y a des contraintes permettant de savoir comment le compilateur va aligner la mémoire, c'est-à-dire connaître le nombre d'octets de bourrage qu'il va ajouter. Il est possible de connaître les contraintes d'alignement d'un type, à l'aide de l'opérateur `_Alignof` défini dans l'en-tête `stdalign.h`. Par exemple le code suivant nous donne les contraintes d'alignement pour les types de notre structure :

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

struct etudiant {
  float taille;
  char genre;
  int age;
};

int main(int argc, char *argv[]) {
  printf("float = %d\n", _Alignof(float));  // affiche 4
  printf("char = %d\n", _Alignof(char));    // affiche 1
  printf("int = %d\n", _Alignof(int));      // affiche 4
  return EXIT_SUCCESS;
}
```

Comme nous pouvons le voir, lorsque le compilateur va considérer le champs `age` il va se rendre compte qu'il y a un problème puisque la taille d'un `int` est 4 et que la RAM utilise des blocs de huit octets. En effet, pour qu'il n'y ait pas de soucis, il est nécessaire qu'un champ de type `int` soit précédé d'un nombre d'octets qui soit multiple de quatre (d'où les trois octets de bourrage pour arriver à 12). Considérons un autre exemple :

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

struct etudiant {
  char genre;
  double age;
};

int main(int argc, char *argv[]) {
  printf("double = %d\n", _Alignof(double));       // affiche 8
  printf("size = %d\n", sizeof(struct etudiant));  // affiche 16
  return EXIT_SUCCESS;
}
```

Comme nous pouvons le voir la taille de la structure est 16 est non 9, puisque un `double` est aligné sur 8 octets. Un dernier petit exemple pour la route :

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

struct etudiant {
  char a, b, c;
};

int main(int argc, char *argv[]) {
  printf("size = %d\n", sizeof(struct etudiant));  // affiche 3
  return EXIT_SUCCESS;
}
```

Puisqu'un `char` doit être aligné sur 1 octet, alors il est normal que la taille de la structure soit 3.

11.3 Les champs de bits

En plus de la déclaration classique des champs d'une structure, il est également possible de leurs spécifier un nombre de bits, appelé « champ de bits ». Pour cela, il suffit de faire suivre le nom du champ par un signe deux-points et d'un nombre représentant le nombre de bits que nous souhaitons affecter au champ. Un champ de bits est interprété comme un entier. Nous pouvons par exemple représenter la date avec un champ de bits :

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

struct date {
  unsigned jour : 5;
  unsigned mois : 4;
  unsigned annee : 15;
};

int main(int argc, char *argv[]) {
  struct date d = {14, 03, 2022};
  printf("%d/%d/%d\n", d.jour, d.mois, d.annee);   // affiche 14/3/2022
  return EXIT_SUCCESS;
}
```

Les champs de bits doivent être suffisamment longs pour contenir le modèle binaire. Par exemple, le code suivant ne compilera pas :

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

struct test {
  char c : 12;
};

int main(int argc, char *argv[]) {return EXIT_SUCCESS;}
```

Les champs de bits ont la même sémantique que le type entier. Cela signifie qu'un champ de bits est utilisé dans les expressions exactement de la même façon qu'une variable du même type de base serait utilisée, quel que soit le nombre de bits dans le champ de bits. Pour illustrer cela reprenons notre exemple avec la date et remplaçons les `unsigned` par des `int` :

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

struct date {
  int jour : 5;
  int mois : 4;
  int annee : 15;
};

int main(int argc, char *argv[]) {
  struct date d = {30, 12, 1777};
  printf("%d/%d/%d\n", d.jour, d.mois, d.annee);  // affiche -2/-4/1777
  return EXIT_SUCCESS;
}
```

Dans l'exemple précédent la valeur affichée est -2 au lieu de 30 attendue. Cela vient du fait que 30 en binaire s'écrit `11110` ce qui correspond bien à -2 en représentation binaire en complément à deux si le nombre de bits utilisés pour représenter notre nombre est 5 (nous pouvons appliquer le même raisonnement pour 12 et -4).
Comme pour les structures sans champs de bits, la taille dépendra des éléments et des contraintes d'alignement. En ce qui concerne la structure `date`, cette dernière aura une taille de 4 octets puisqu'elle comporte 3 champs dont la somme est égale à 24.
Il faut aussi noter qu'il n'est pas possible d'avoir de pointeurs vers des membres de champ de bits car ils peuvent ne pas commencer à une limite d'octet.

11.4 Les unions

Une union est comme une structure, c'est-à-dire un regroupement d'objet de type différents. Cependant, contrairement au structure, une union est un agrégat qui ne peut contenir qu'un seul de ses membres à la fois. Plus précisément, une variable de type union stocke l'une des valeurs définies par ce type. Les

mêmes règles gouvernent les déclarations de structure et d'union et elles peuvent également avoir des champs de bits. Considérons le cas où nous souhaitons représenter un nombre avec différentes tailles :

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

union nombre {
  char int8;
  int int32;
  long int64;
};

int main(int argc, char *argv[]) {
  return EXIT_SUCCESS;
}
```

En ce qui concerne la taille d'une union il est toujours possible d'utiliser l'opérateur `sizeof`. Le nombre octets que représente une union correspond à la taille du plus grand type stocké. Pour ce qui est de l'exemple précédent `sizeof(union nombre)` retournera 8 (la taille d'un `long`). Puisque la taille est 8 nous voyons très bien qu'il n'est pas possible de stocker tous les types de l'union à la fois! Notons que comme les structures, les unions peuvent contenir des bits de bourrages (ils seront placés à la fin).
L'accès aux membres est réalisé de la même manière que pour les structures. Pour ce qui est de l'initialisation, elle diffère de l'initialisation des structures puisque nous n'avons qu'un champ à remplir. Il est important de noter que le compilateur effectuera bien les opérations sur les différents champs en fonction du type qui sera sélectionné. Voyons un peu ce qu'il se passe quand nous considérons l'opération de décalage de bits lorsque notre union comporte un entier signé et un entier non signé.

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

union nombre {
  int int32;
  unsigned u_int32;
};

int main(int argc, char *argv[]) {
  union nombre n = {-1};
  printf("%d %u\n", n.int32, n.u_int32);           // affiche -1 4294967295
  printf("%d %u\n", n.int32 >> 1, n.u_int32 >> 1); // affiche -1 2147483647
  printf("%u %d\n", n.int32 >> 1, n.u_int32 >> 1); // affiche 4294967295 2147483647
  return EXIT_SUCCESS;
}
```

Dans l'exemple précédent, nous voyons très bien que lorsque nous décalons les bits de notre union `n` d'un rang vers la droite nous avons un comportement différent en fonction de ce que nous voyons `n` comme un entier signé ou un entier non signé. Lorsque nous voyons `n` comme un entier signé alors c'est un 1 qui est inséré à gauche (puisque notre nombre est négatif). Dans le cas où `n` est vue comme un entier nom signé alors c'est un 0 qui est inséré à gauche.

11.5 Structures et unions non nommées

Comme permit par la norme ISO C11 et pour des raisons de compatibilité, `gcc` permet de définir des structures et des unions sans noms. Par exemple :

```c
struct test {
  int a;
  union {
    int b;
    float c;
  };
  int d;
};
```

Cette définition permet `b` comme un entier signé et `c` comme un `float` tout en allouant uniquement 4 octets pour stocker cette information. Ce genre de définition peut être utile dans le cas où nous avons des opérations à réaliser sur notre structure et qu'en fonction de la situation nous souhaitons que notre variable soit d'un type plutôt qu'un autre. Cette notation sans nom à l'avantage de considérer les champs de l'union comme s'ils étaient des champs de la structure, ce qui implique qu'ils sont donc accessibles comme tel. Dans l'exemple suivant nous initialisons la structure `test` de telle manière que l'union reçoit 1.

```
#include <stdio.h>
#include <stdlib.h>
struct test {
  int a;
  union {
    int b;
    float c;
  };
  int d;
};

int main(int argc, char *argv[]) {
  struct test t = {1, 1, 2};
  printf("%d %f\n", t.b, t.c);   // affiche 1 0.00000
  return EXIT_SUCCESS;
}
```

Nous remarquons dans l'exemple précédent que nous avons les même règles que pour les unions, c'est-à-dire que l'affectation dépend du type que nous allons utilisé (ici un entier). Nous aurions aussi bien pu nommer l'union, mais dans ce cas il aurait nécessaire de dépointer une fois de plus pour accéder au champ voulu.

```
#include <stdio.h>
#include <stdlib.h>
struct test {
  int a;
  union {
    int b;
    float c;
  } u;
  int d;
};

int main(int argc, char *argv[]) {
  struct test t = {1, 1, 2};
  printf("%d %f\n", t.u.b, t.u.c);   // affiche 1 0.00000
  return EXIT_SUCCESS;
}
```

Il aurait tout à fait été possible d'utiliser un `cast` afin d'avoir le même comportement (en fait nous pouvons toujours nous en sortir sans union . . .mais il faut faire un peu de gymnastique). Cependant, comme le montre l'exemple suivant utiliser un tel procédé permet de rendre notre code plus clair :

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

struct arbre {
  int val;
  struct arbre *g, *d;
  union {
    int compteur;
    char flag;
  };
};

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

```
14   struct arbre *a;
15
16   /* du codes ... */
17
18   if (a->flag) return EXIT_FAILURE;
19   return EXIT_SUCCESS;
20 }
```

Dans l'exemple précédent, bien que nous aurions pu utiliser le champ `compteur` pour représenter notre `flag`, il est évident que d'utiliser `compteur` dans la condition ligne 18 aurait été largement moins lisible.

11.6 Les définitions de types par `typedef`

Il est tout à fait possible de définir un type (plus précisément de créer un alias) à partir d'un type existant. La syntaxe pour définir un type est similaire à une déclaration de variable, si ce n'est que celle-ci doit être précédée du mot-clé `typedef`. Dans ce cas, l'identificateur choisi désignera un type et non une variable. Prenons par exemple le cas où nous souhaitons créer un type `boolean` :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef int boolean;
5
6  int main(int argc, char *argv[]) {
7    boolean b = 1;
8    return EXIT_SUCCESS;
9  }
```

Utiliser des alias dans votre code n'est pas une obligation, mais cela améliore grandement la lisibilité de votre code en nommant correctement les objets que vous manipulez. De plus, il est possible d'utiliser la définition de type sur des objets plus complexes comme les `enum`, `struct` et `union`. Par exemple, au lieu de considérer un entier pour représenter un booléen nous pouvons aussi utiliser un type énuméré.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef enum { FAUX, VRAI } boolean;
5
6  int main(int argc, char *argv[]) {
7    boolean b = VRAI;
8    return EXIT_SUCCESS;
9  }
```

Dans l'exemple précédent nous avons réalisé en une opération la déclaration du type énuméré et la définition du type `boolean`. Il est aussi possible de faire cela en deux étapes, mais dans ce cas il faudra nommer le type énuméré.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  enum _boolean { FAUX, VRAI };
5  typedef enum _boolean boolean;
6
7  int main(int argc, char *argv[]) {
8    boolean b = VRAI;
9    return EXIT_SUCCESS;
10 }
```

En ce qui concerne les structures récursives, comme les listes, il est dans tous les cas nécessaire de nommer la structure afin de l'utiliser comme un champ.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
```

```c
typedef struct _list {
  int val;
  struct _list *next;
} list;

int main(int argc, char *argv[]) {
  list *l = NULL;
  return EXIT_SUCCESS;
}
```

Comme pour une déclaration de variable, vous pouvez déclarer plusieurs alias en une déclaration.

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

typedef struct _list {
  int val;
  struct _list *next;
} list, *ptr_list;

int main(int argc, char *argv[]) {
  ptr_list l = NULL;
  return EXIT_SUCCESS;
}
```

Chapitre 12

Les entrées-sorties

Chapitre 13

Les directives au préprocesseur

Chapitre 14

La programmation modulaire