Programmation en C

Benjamin Godard

Table des matières

Chapitre 1

Bases du C

1.1 Généralités

1.1.1 Historique

- Le C, développé en 1971 par Dennis Ritchie, est l'un des langages les plus utilisés (www.developpez.com/).
- Le C a de nombreux avantages : proche du langage machine, rapide, portable, et très documenté.
- Il est à l'origine de langages plus évolués et très utiles : Objective-C, C++ (programmation orientée objet).
- Il est baptisé "langage de programmation système" car il sert à l'écriture de compilateurs et de systèmes d'exploitations : UNIX, OS-X, Windows.

1.1.2 Normes

Une norme définit toutes les règles de programmation d'un langage (e.g. types de variables, de fonctionnalités, déclarations, ...). Le C possède à l'heure actuelle plusieurs normes dont l'accès dépend du compilateur utilisé.

NORME	INSTITUT	GCC	AMÉLIORATIONS
C89	ANSI[1]	✓	description des arguments dans les fonctions, syntaxe modifiée, affectation des structures, énumération, redéfinition de la bibliothèque standard
C99	ISO[2]	✓	tableaux à taille variable, pointeurs restreints, complexes, type long long, syntaxe des commentaires C++, fonctions inline, déclarations / instructions
C11	ISO[2]	✗	programmation multi-thread, programmation générique

[1] American National Standards Institute

[2] International Organization for Standardization

1.1.3 Références

- Kernighan & Ritchie, **Le langage C, Norme ANSI - 2nd Edition**
- Delannoy, **Programmer en langage C**
- Delannoy, **Exercices en langage C**
- Le site du zéro, Nebra, https://openclassrooms.com/
- Club des développeurs, http://c.developpez.com

1.2 Structure d'un programme

Exemple

```
/*************************************************************************/
/* Cet exemple illustre la structure d'un programme typique écrit en C */
/*************************************************************************/
#include <stdio.h>

#define NFILE 100
#define NCHAR 50

const double Wien = 2.898e-3 ;

void viderbuffer(void) ;
int count_lines(char *) ;

/* Fonction principale - doit figurer dans tout programme C ! */
int main (void)
{• • •}

/* Fonction viderbuffer */
void viderbuffer(void)
{• • •}

/* Fonction count_lines : compte les lignes d'un fichier */
int count_lines(char *name)
{• • •}
```

Exemple

```
int main (void)
{
    char filename[NFILE] ;
    int nlamb ;

    printf("Nom du fichier contenant les données ?\n") ;
    scanf("%s",filename) ;
    viderbuffer() ;

    nlamb = count_lines(filename) ;
    printf("Nombre de lignes lues : %d\n",nlamb) ;

    return 0 ;
}
```

Deux remarques importantes :

- les commentaires sont encadrés par **/*** et ***/** et peuvent s'étendre sur plusieurs lignes
- tout programme est composé d'instructions. Les { } rassemblent des groupes d'instructions. Au sein de ces groupes, chaque instruction se termine par "**;**".

1.2.1 Directives de précompilation

Les directives de précompilation permettent de modifier le code avant sa compilation. Elles commencent par un #.

```
#include <nom_du_fichier>
#include "nom_du_fichier"
```

permettent d'inclure des fichiers dans le code. "" signifie que le fichier se trouve dans le répertoire courant. <> signifie qu'il faut chercher le fichier dans les répertoires connus du compilateur (e.g. /usr/include).

```
#define NOM expression
#undef  NOM
```

permettent de définir une constante symbolique ou une macro `NOM` et d'annuler sa définition.

```
#ifdef NOM
instructions 1
#else
instructions 2
#endif
```

permettent d'inclure du code de manière conditionnelle (si `NOM` est défini) au moment de la compilation

1.2.2 Variables globales, prototypes de fonction

- Les variables définies en dehors des fonctions sont **globales**, i.e. accessibles par toutes les fonctions suivantes.
- Un **prototype de fonction** indique le type retourné par la fonction ainsi que les types de ses arguments. Une fonction ainsi déclarée peut-être appelée par toutes les fonctions suivantes.

1.2.3 Les fonctions

- La fonction `main` est la fonction principale du programme, i.e. celle exécutée en premier. **Elle est obligatoire** !
- Toute fonction utilise des arguments et renvoie une valeur. Ces caractéristiques sont indiquées dans le prototype de la fonction et lors de sa définition. Si aucun argument ou aucune grandeur retournée, utiliser `void`.
- En C, tous les arguments des fonctions se passent **par valeur**. Celle-ci est copiée dans une nouvelle variable, **locale à la fonction**.
- La seule manière pour une fonction appelée de modifier une variable de la fonction appelante est de transmettre son adresse mémoire, i.e. **utiliser un pointeur**.

1.3 Compilation

La compilation consiste à traduire un programme écrit dans un langage en langage machine binaire exécutable. Le compilateur standard en version libre du langage C est gcc (GNU Compiler Collection).

```
gcc -options -o prog prog.c
```

1.3.1 Etapes de compilation

La compilation d'un programme C s'effectue en réalité en 4 étapes successives qui peuvent être décomposées via

```
gcc -E prog.c > prog.i
gcc -S prog.i > prog.s
gcc -c prog.s > prog.o
gcc -o prog prog.o
```

1. Preprocessing : supprime les commentaires ; étend et inclut les fichiers d'en-tête dans le programme source ; interprète les directives de compilation # et supprime les parties non compilées ; remplace les macros dans tout le code ; vérifie la syntaxe et les déclarations de fonctions.
2. Compilation : transforme le code C en Assembleur, code machine contenant des instructions qui manipulent la mémoire et le processeur directement. Les erreurs d'initialisation de mémoire sont détectées ici.
3. Assembling : produit le code dit "objet", traduction des instructions du code Assembleur en binaire. Le fichier obtenu contient des trous correspondants aux appels de fonctions dans d'autres fichiers C ou bibliothèques.
4. Linking : édite les liens ; réunit tous les fichiers objets et les fonctions des bibliothèques en un code binaire unique exécutable.

Les fichiers binaires produits au cours de la procédure (prog.o et prog) peuvent être lus avec

```
od -x prog.o
```

1.3.2 Options du compilateur `gcc`

`gcc` comprend des centaines d'options de compilation dont les descriptions sont accessibles sur le site https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html, comme par exemple

OPTION	EFFET
`-ansi`	permet d'être conforme à la norme C89 en éteignant certaines propriétés de gcc qui sont incompatibles avec C89 (reconnaissance des commentaires écrits en C++, macros prédéfinis, ...)
`-Wall`	allume un ensemble d'options d'avertissement et de vérifications (e.g. variables ou fonctions implicites, labels / variables non utilisées, brackets manquants, ...)
`-Wextra`	allume des options d'avertissement non allumées par Wall (e.g. paramètres non utilisés, initialisations manquantes, ...)

1.3.3 Autres compilateurs

Même si `gcc` est reconnu comme étant le compilateur standard, il existe des dizaines de compilateurs différents, développés en licence libre ou privée, et adaptés / optimisés pour différents systèmes d'exploitation et processeurs.

COMPILATEUR	SYSTÈME	LICENCE	REMARQUES
Microsoft Visual Studio Express	Windows	Free Version	Version pédagogique d'un compilateur industriel
Tiny C Compiler	GNU/Linux, Windows	LGPL	
Clang	GNU/Linux, Windows, UNIX, OSX	University of Illinois, NCSA License	Compilateur C utilisant LLVM pour compiler de l'Objective C et du C++
GNU C Compiler	GNU/Linux, Windows, UNIX, OSX	GPL	Le standard. Installé sur la plupart des systèmes UNIX

Table des matières

Chapitre 2

Variables, opérateurs et librairies

2.1 Variables

2.1.1 Types et déclaration

Pour déclarer une variable dans un programme, il faut indiquer son type suivi de son nom,

```
type nom ;
```

Il existe 7 types de variables en C en C89. Booléens et complexes ne sont définis que depuis C99. Les tailles et bornes des types de base dépendent de la machine. Le tableau suivant donne ces informations pour un processeur de 64 bits.

Pour construire un code portable, il est recommandé d'utiliser les appellations alternatives (accessibles via le fichier d'en-tête **<stdlib.h>**) qui sont explicites et invariantes d'un processeur à l'autre.

TYPE	ALTERNATIVE	TAILLE	VAL. MIN	|VAL. MIN|	VAL. MAX
char	int8_t	1 octet	-2^7		2^7-1
short	int16_t	2 octets	-2^{15}		$2^{15}-1$
int	int32_t	4 octets	-2^{31}		$2^{31}-1$
long	int64_t	8 octets	-2^{63}		$2^{63}-1$
float		4 octets	-3.4×10^{38}	1.2×10^{-38}	3.4×10^{38}
double		8 octets	-1.8×10^{308}	2.2×10^{-308}	1.8×10^{308}
long double		16 octets	-1.1×10^{4932}	3.4×10^{-4932}	1.1×10^{4932}

2.1.2 Nom des variables

Le C fait la différence entre majuscules et minuscules. Pour éviter les confusions, il est recommandé d'utiliser les minuscules pour les variables et les majuscules pour les constantes symboliques.

Les noms peuvent contenir des lettres (sans accent), chiffres et le caractère '_' et ne peuvent commencer par un chiffre. Il est également déconseillé de commencer une variable par '_'. Il existe en outre un certain nombre de noms réservés dont la liste complète est accessible à l'adresse suivante
http://www.gnu.org/software/libc/manual/html_node/Reserved-Names.html.

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

2.1.3 Attributs / modificateurs

Il est possible d'associer des attributs aux variables (lors de leur déclaration) qui modifient leur sens et leur portée.

```
attribut type nom ;
```

ATTRIBUT	SIGNIFICATION
signed	impose que la variable soit positive (entiers seulement)
unsigned	ne présume pas du signe de la variable (entiers seulement)
const	la variable ne peut être modifiée au cours du programme
extern	déclare une variable globale. Celle-ci doit être définie ailleurs
static	rend la variable persistante ; limite sa portée au fichier
register	demande au compilateur un accès rapide pour cette variable

2.1.4 Initialisation

- Les variables globales doivent être initialisées par une expression constante au moment de leur définition.
- Les variables locales peuvent être initialisée n'importe où et par une expression non constante.
- Si non initialisées, les variables peuvent prendre n'importe quelle valeur. Il est donc fortement recommandé d'**initialiser toute variable lors de sa déclaration** par une expression constante.

2.2 Constantes

2.2.1 Constantes littérales

Les constantes littérales sont les valeurs numériques ou caractères entrés en dur dans le code source.

Exemple

```
int i = 2 ;                          /* 2 : constante entière */
double kb = 1.38065e-23 ;            /* 1.38065e-23 : constante flottante */
char c = 'A' ;                       /* 'A' : constante caractère */
char chaine[8] = "Bonjour" ;         /* "Bonjour" : constante de type chaîne */
```

2.2.2 Constantes symboliques

Il en existe de deux sortes : les variables déclarées avec le modificateur **const**

```
const type nom ;
```

et les constantes dites "préprocesseur", traitées à la précompilation et définies par (notez l'absence de ' ;')

```
#define NOM constante
```

Les constantes préprocesseur permettent de définir des constantes réelles ou entières, mais aussi des macros.

Exemple

```
#define PI 3.14159
#define NMAX 100
#define max(A,B) ((A) > (B) ? (A) : (B))
```

2.3 Opérateurs

Les opérateurs sont des traitements appliqués à des variables. Le C comprend 45 opérateurs différents qui s'appliquent sur une, deux ou trois variables, et qui peuvent être combinés pour former des expressions complexes.

2.3.1 Opérateurs arithmétiques

Ceux-ci s'utilisent de la manière suivante

```
a op b
```

OPÉRATEUR	SIGNIFICATION
+	addition
-	soustraction
*	multiplication
/	division
%	modulo : reste de la division de a par b

2.3.2 Opérateurs de comparaison et logiques

Ils s'utilisent de la même manière que les opérateurs arithmétiques. Les expressions reliées par && ou || sont évaluées de gauche à droite, et l'évaluation cesse dès que la véracité ou la fausseté du résultat est établie.

OPÉRATEUR	SIGNIFICATION
>	supérieur
>=	supérieur ou égal
<	inférieur
<=	inférieur ou égal
==	égal
!=	différent
&&	et
||	ou

2.3.3 Opérateurs d'affectation

Ils s'utilisent de la même manière que les opérateurs arithmétiques. Le C associe à la plupart des opérateurs binaires **op**, un opérateur d'affectation de la forme **op=**.

OPÉRATEUR	SIGNIFICATION
=	affectation de b à a
+=	équivalent de `a = a + b`
-=	équivalent de `a = a - b`
*=	équivalent de `a = a * b`
...	

2.3.4 Opérateurs de conversion

Le C permet de forcer une conversion de type dans toute expression, via l'opérateur unaire **cast**, qui s'utilise de la manière suivante

```
(nom_de_type) a
```

Exemple

```
int n = 3 ;
double x = 0.0 ;
x = sqrt( (double) n) ;
```

note : le résultat d'une opération entre différents types se fait dans le type capable de stocker le plus grand nombre.

2.3.5 Opérateurs d'incrémentation

Le C comporte deux opérateurs unaires, ++ et −−, qui servent à incrémenter et décrémenter les variables entières. En fonction de la position de l'opérateur, l'incrémentation est effectuée avant ou après l'expression.

Exemple

```
int i = 0, j = 0 ;
i = 1 ;
j = i++ ;       /* effectue d'abord j = i et ensuite i = i + 1. On a alors j = 1 et i = 2 */

i = 1 ;
j = ++i ;       /* effectue d'abord i = i + 1 et ensuite j = i. On a alors j = 2 et i = 2 */
```

2.3.6 Opérateurs de traitement de bits

Le C inclut des opérateurs qui permettent de manipuler des variables de type `char`, `short`, `int` et `long`, au niveau des bits eux-mêmes.

OPÉRATEUR	SIGNIFICATION
&	et bit à bit
|	ou inclusif bit à bit
^	ou exclusif bit à bit
<<	décalage à gauche
>>	décalage à droite

Exemple

```
int n = 12, m = 7 ;
n = n & 0177 ;      /* met à 0 les bits de n qui ne font pas partie de ses 7 bits de poids faible*/
n = n & m ;         /* met à 1 tous les bits de n qui sont à 1 dans n et dans m*/
n = n | m ;         /* met à 1 tous les bits de n qui sont à 1 dans n ou dans m*/
```

2.3.7 Priorités et ordre d'évaluation

La table suivante récapitule les règles de priorité et d'associativité des opérateurs. Les opérateurs sur une même ligne ont le même degré de priorité, supérieur à celui de la ligne suivante. En cas de doute, toujours utiliser des parenthèses.

OPÉRATEUR	TYPE	ASSOCIATIVITÉ
() [] -> .	unaire	de gauche à droite
! ~ ++ −− + - * & (type) sizeof	unaire	de droite à gauche
* / %	binaire	de gauche à droite
+ -	binaire	de gauche à droite
« »	binaire	de gauche à droite

< <= > >=	binaire	de gauche à droite
== !=	binaire	de gauche à droite
&	binaire	de gauche à droite
^	binaire	de gauche à droite
|	binaire	de gauche à droite
&&	binaire	de gauche à droite
||	binaire	de gauche à droite
? :	ternaire	de droite à gauche
= += -= *= /= %= &= ^= |= <<= >>=	binaire	de droite à gauche
,	binaire	de gauche à droite

Attention, le nombre d'opérateurs et le fait que ceux-ci peuvent se combiner à loisir permettent d'écrire des instruction condensées. Celles-ci sont non seulement difficiles à lire mais aussi difficile à déboguer car "rien" n'est interdit.

Exemple

```
int x = 0 ;

if (x == 3)      /* test si x == 3*/
{...}            /* si oui, effectue les instructions entre {}*/

if (x = 3)       /* affecte 3 à x puis teste x. Ceci est toujours vrai */
{...}            /* les instructions entre {} sont nécessairement effectuées*/
```

Exemple

Evaluez l'expression suivante. Pourquoi la variable i n'est-elle pas incrémentée ?

```
double x = 1.0, y = 1.0 ;
int i = 1, j = 2, k = 4, l = 4 ;

if ( x *= y || k % l & 177 && ++i != j )
{...}
```

2.4 Fichiers d'en-tête & librairies standards

Les fonctions, types et macros de la bibliothèque standard sont déclarés dans les fichiers d'en-tête standards.

`<assert.h>`	`<float.h>`	`<math.h>`	`<stdarg.h>`	`<stdlib.h>`
`<ctype.h>`	`<limits.h>`	`<setjmp.h>`	`<stddef.h>`	`<string.h>`
`<errno.h>`	`<locale.h>`	`<signal.h>`	`<stdio.h>`	`<time.h>`

Pour utiliser un fichier d'en-tête, écrire dans le programme :

```
1 #include <fichier d'en-tete>
```

Pour lier les fonctions d'une librairie (e.g. libname.so) au programme, compiler ainsi :

```
1 gcc -o mon_prog.exe mon_prog.c -lname -Lrepertoire
```

2.4.1 Les entrées sorties : <stdio.h>

Les fonctions, types et macros de cet en-tête représentent un tiers de la bibliothèque et incluent, les opérations sur les fichiers, la lecture et l'écriture de variables et la gestion d'erreurs. La librairie associée est **libc.so** et est liée par défaut.

- `FILE *fopen (const char *filename, const char *mode)` — ouvre le fichier indiqué et retourne un flot ou NULL si la tentative échoue
- `FILE *fclose (FILE *stream)` — force l'écriture des données non écrites du flot, efface le contenu des tampons et ferme le flot. Retourne EOF en cas d'erreur
- `int rename (const char *oldname, const char *newname)` — change le nom du fichier `oldname` en `newname`. Retourne une valeur $\neq 0$ en cas d'échec
- `FILE *tmpfile (void)` — crée un fichier temporaire qui sera automatiquement détruit à sa fermeture ou lors de la fin normale du programme
- `int fprintf(FILE *stream, const char *format, ...)` — convertit ses arguments selon le `format` donné et écrit le résultat sur le flot `stream`. Retourne une valeur négative en cas d'erreur
- `int printf(const char *format, ...)` — équivaut à `fprintf(stdout, ...)`
- `int fscanf(FILE *stream, const char *format, ...)` — lit les données du flot `stream` selon le `format` donné et affecte les valeurs converties aux arguments suivants, chacun devant être un pointeur. Retourne EOF si une erreur s'est produite
- `int scanf(const char *format, ...)` — équivaut à `fscanf(stdin, ...)`
- `int fgetc(FILE *stream)` — retourne le caractère suivant du flot `stream`, ou EOF si la fin du fichier est atteinte ou si une erreur survient
- `int *fgets(char *s, int n, FILE *stream)` — lit les `n-1` caractères suivant du flot (où jusqu'à atteindre un retour à la ligne) et les place dans le tableau `s`. Retourne NULL si la fin du fichier est atteinte ou si une erreur survient
- `int getchar(void)` — équivaut à `fgetc(stdin)`
- `int fputc(int c, FILE *stream)` — écrit le caractère `c` sur le flot `stream`. Retourne le caractère écrit ou EOF en cas d'erreur
- `int *fputs(const char *s, FILE *stream)` — écrit la chaîne de caractères sur le flot `stream`. Retourne EOF en cas d'erreur
- `int putchar(void)` — équivaut à `fputc(c,stdout)`

2.4.2 Tests sur un caractère : <ctype.h>

Les fonctions des cet en-tête visent à tester les caractères. La valeur retournée est non nulle si vrai, zéro sinon. La librairie est liée par défaut.

- `int iscntrl (char c)` — test si `c` est un caractère de contrôle
- `int isdigit (char c)` — test si `c` est un chiffre décimal
- `int isgraph (char c)` — test si `c` est un caractère imprimable, espace non compris
- `int isprint (char c)` — test si `c` est un caractère imprimable, espace compris
- `int islower (char c)` — test si `c` est une lettre minuscule
- `int isupper (char c)` — test si `c` est une lettre majuscule
- `int isspace (char c)` — test si `c` est un espace, fin de ligne, retour chariot ou tabulation
- `int tolower (char c)` — convertit `c` en minuscules
- `int toupper (char c)` — convertit `c` en majuscules

2.4.3 Traitement des chaînes de caractères : <string.h>

Les fonctions contenues ici permettent le traitement, la manipulation et la comparaison de chaînes de caractères.

- `char *strcpy (char *s, const char ct)` — copie la chaine `ct`, y compris '\0' dans la chaîne `s` ; retourne `s`
- `char *strncpy (char *s, const char ct, size_t n)` — copie au plus `n` caractères de la chaîne `ct` dans `s` ; complète par des '\0' si nécessaire
- `char *strcat (char *s, const char ct)` — concatène la chaîne de `ct` à la suite de la chaîne `s` ; retourne `s`
- `char *strncat (char *s, const char ct, size_t n)` — concatène au plus `n` caractères de la chaîne de `ct` à la suite de la chaîne `s` ; termine `s` par '\0' ; retourne `s`
- `int strcmp (const char cs, const char ct)` — compare la chaîne `cs` à la chaîne `ct` ; retourne une valeur négative si `cs` < `ct`, nulle si `cs` == `ct`, positive si `cs` > `ct`
- `int strncmp (const char cs, const char ct, size_t n)` — compare au plus `n` caractères de la chaîne `cs` à `ct` ; retourne une valeur négative si `cs` < `ct`, nulle si `cs` == `ct`, positive si `cs` > `ct`
- `char *strchr (const char cs, char c)` — retourne un pointeur sur la première occurrence de `c` dans `cs`, ou NULL si `c` ne figure pas dans `cs`
- `char *strrchr (const char cs, char c)` — retourne un pointeur sur la dernière occurrence de `c` dans `cs`, ou NULL si `c` ne figure pas dans `cs`
- `char *strstr (const char cs, const char ct)` — retourne un pointeur sur la première occurrence de la chaîne `ct` dans `cs`, ou NULL si `ct` ne figure pas dans `cs`
- `size_t strlen (const char cs)` — retourne la longueur de la chaîne `cs`

2.4.4 Fonctions mathématiques : <math.h>

Ce fichier d'en-tête contient les prototypes de nombreuses fonctions mathématiques de base. Toutes ces fonctions sont définies dans la librairie **libm.so**.

- `double sin (double x)` — sinus de x
- `double cos (double x)` — cosinus de x
- `double tan (double x)` — tangente de x
- `double asin (double x)` — arc sinus de x, dans l'intervalle [$-\pi/2$, $\pi/2$]
- `double acos (double x)` — arc cosinus de x, dans l'intervalle [0, π]
- `double atan (double x)` — arc cosinus de x, dans l'intervalle [$-\pi/2$, $\pi/2$]
- `double atan2 (double x, double y)` — arc cosinus de x/y, dans l'intervalle [$-\pi$, π]
- `double sinh (double x)` — sinus hyperbolique de x
- `double cosh (double x)` — cosinus hyperbolique de x
- `double tanh (double x)` — tangente hyperbolique de x
- `double exp (double x)` — fonction exponentielle e^x
- `double log (double x)` — logarithme népérien $\ln(x)$
- `double log10 (double x)` — logarithme à base 10 $\log_{10}(x)$
- `double pow (double x, double y)` — x^y. Erreur si $x = 0$ et $y \leqslant 0$ ou $x < 0$ et y non entier
- `double sqrt (double x)` — $\sqrt{x}$
- `double ceil (double x)` — plus petit entier supérieur ou égal à x, exprimé en double
- `double floor (double x)` — plus grand entier inférieur ou égal à x, exprimé en double
- `double fabs (double x)` — valeur absolue $|x|$
- `double ldexp (double x, int n)` — $x \times 2^n$
- `double modf (double x, double *ip)` — sépare x en parties entière et fractionnaire, du même signe que x. Met la partie entière dans `*ip` et retourne la partie fractionnaire
- `double fmod (double x, double y)` — reste de x/y, exprimé en double

Table des matières

Chapitre 3

Entrées et sorties

La lecture des données depuis le clavier ou un fichier et leur écriture à l'écran ou dans un fichier s'effectue à l'aide des variables, macros et fonctions déclarées dans <stdio.h>.

3.1 Gestion des flux de données

3.1.1 Flux de données

Un **flot** ou **flux de données** est une source ou une destination de données, texte ou binaires. En mode texte, un flot est une suite de lignes, séparées les unes des autres par le caractère '\n', qui prend fin au caractère EOF. Quand l'en-tête <stdio.h> est utilisé, trois flots sont automatiquement créés, associés aux variables globales

- `stdin` → entrée standard (clavier)
- `stdout` → sortie standard (écran)
- `stderr` → sortie erreur (écran)

<stdio.h> permet de définir de nouvelles variables, **pointeurs de fichiers**, que l'on peut associer à des flots

```
1 FILE *nom_flot ;
```

Ces pointeurs dovent toujours être initialisés à NULL (adresse invalide). Chaque flot possède un **tampon** associé.

3.1.2 Manipulation de fichiers

On associe un flot à un fichier ou un périphérique en l'ouvrant. Si l'ouverture échoue, la fonction `fopen` renvoie NULL.

```
1 nom_flot = fopen ("nom_de_fichier", "mode") ;
```

Mode	Action	Curseur	Conditions
"r"	lecture seule	début de fichier	le fichier doit exister
"w"	écriture seule	début de fichier	écrase le contenu
"a"	écriture seule	fin de fichier	
"r+"	écriture & lecture	début de fichier	le fichier doit exister
"w+"	écriture & lecture	début de fichier	écrase le contenu
"a+"	écriture & lecture	fin de fichier	

On annule le lien en fermant le flot. Le pointeur doit alors toujours être réaffecté à NULL (adresse invalide).

```
1 fclose (nom_flot) ;
```

3.2 Écriture & lecture

3.2.1 Séquences d'échappement

Les séquences d'échappement sont des caractères au comportement spécial

SÉQUENCE	EFFET	SÉQUENCE	EFFET
\a	alerte, sonnerie	\v	tab. verticale
\b	retour en arrière	\\	affiche un \
\f	saut de page	\?	affiche un ?
\n	fin de ligne	\!	affiche un !
\r	retour chariot	\'	affiche un '
\t	tab. horizontale	\"	affiche un "

3.2.2 Écriture avec fprintf

```
1 fprintf(nom_flot, "format", arg1, arg2, ...) ;
```

permet d'afficher la chaîne de caractères `"format"` dans le flot `nom_flot`.

• `printf("format", arg1, arg2, ...)` est équivalente à `fprintf(stdout, "format", arg1, arg2, ...)`.

• `"format"` peut contenir : des caractères ordinaires et des **spécifications de conversion** commençant par % et finissant par un **caractère de conversion**. Celles-ci provoquent l'impression de l'un des arguments suivants de `fprintf`.

SPÉCIF.	TYPE	NOTATION	SPÉCIF.	TYPE	NOTATION
%hd	short	décimale	%c	char	un seul caractère
%d	int	décimale	%s	*char	chaîne de caractères
%ld	long	décimale	%f	float	point décimal
%ho	short	octale	%lf	double	point décimal
%o	int	octale	%Lf	long double	point décimal
%lo	long	octale	%e	float	exponentielle
%hx	short	hexadécimale	%le	double	exponentielle
%x	int	hexadécimale	%Le	long double	exponentielle
%lx	long	hexadécimale			

Note 1 : si l'on écrit un `char` avec un format décimal, on affiche la valeur du code ASCII du caractère.

Note 2 : toujours utiliser \n, sinon le résultat de fprintf n'est pas forcément affiché au moment de la commande.

Exemple

```
#include <stdio.h>
int main (void)
{
    int i = 3, j = 4 ;
    float x = 3.45 ;

    printf("%d\n", i) ;              /* Affiche '3' */
    printf("i=%d\n", i) ;            /* Affiche 'i=3' */
    printf("i=%d,j=%d\n", i, j) ;    /* Affiche 'i=3,j=4' */
    printf("i=%5d\n", i) ;           /* Affiche 'i=  3' */
    printf("i=%05d\n", i) ;          /* Affiche 'i=00003' */
    printf("x=%f\n", x) ;            /* Affiche 'x=3.450000' */
    printf("x=%5.2f\n", x) ;         /* Affiche 'x= 3.45' */
    return 0 ;
}
```

3.2.3 Lecture avec fscanf

```
fscanf(nom_flot, "format", &arg1, &arg2, ...) ;
```

permet de lire les données du flot `nom_flot` d'après les spécifications contenues dans `"format"`.

- les "&" signifient que l'on transmet l'adresse des variables `arg1`, `arg2`, ..., et non pas leurs valeurs.
- `scanf("format", &arg1, &arg2, ...)` est équivalente à `fscanf(stdin, "format", &arg1, &arg2, ...)`.
- l'utilisation de `"format"` est similaire à `fprintf` mais pas équivalente.
  - Les espaces et les caractères de tabulation sont ignorés.
  - Seule la largeur maximale d'un champ peut être spécifiée.
  - La lecture d'une chaîne de caractères (%s) s'arrête aux premier caractère d'espacement (" ", \n, \t, \v, \r, et \f).

Exemple

```
#include <stdio.h>
int main (void)
{
    int a = 0 ;
    scanf("%d",&a) ;
    return 0 ;
}
```

Exemple

```
#include <stdio.h>
int main (void)
{
    FILE *fich = NULL ;
    float m_mercure = 0.0, m_venus = 0.0, m_terre = 0.0, m_mars = 0.0 ;
    char nom[10] = "", separateur = '\0' ;

    /* Ouverture en lecture seule du fichier donnees1.txt */
    fich = fopen("donnees1.dat", "r") ;
    if (fich == NULL)
    {
        printf("Erreur à l'ouverture de donnees1.txt\n") ;
        return 1 ;
    }

    fscanf(fich, "%9s %c %f", nom, &separateur, &m_mercure) ;
    fscanf(fich, "%9s %c %f", nom, &separateur, &m_venus) ;
    fscanf(fich, "%9s %c %f", nom, &separateur, &m_terre) ;
    fscanf(fich, "%9s %c %f", nom, &separateur, &m_mars) ;

    fclose(fich) ;
    fich = NULL ;

    return 0 ;
}
```

3.3 Le cas des chaînes de caractères

Une chaîne est un tableau de caractères qui se termine par '\0'. '\0' est nécessaire pour utiliser la spécification %s.

3.3.1 Lecture des blancs : fonction fgets

Comme dit précédemment, la fonction **scanf** arrête la lecture au premier caractère d'espacement rencontré.

Exemple

```
char tab[100] = "";
scanf("%s", tab);          /* Si l'utilisateur saisit "Hello world!" */
printf("%s\n", tab);       /* Le programme affiche "Hello" */
```

La fonction **fgets** permet de résoudre ce problème. Celle-ci lit une chaîne de caractères contenant des blancs dans le flot `nom_flot`, d'une longueur maximale `nbchar`, jusqu'à trouver une fin de ligne (\n) ou une fin de fichier (EOF).

```
1 fgets(nom_chaine, nbchar, nom_flot) ;
```

L'équivalent de **fgets** en écriture est la fonction **fputs**. Celle-ci écrit une chaîne de caractères dans le flot `nom_flot`.

```
1 fputs(nom_chaine, nom_flot) ;
```

Exemple

```
#include <stdio.h>
int main (void)
{
    FILE *fich1 = NULL, *fich2 = NULL;
    char ligne[100] = "";

    fich1 = fopen("donnees1.dat", "r"); /* Ouverture en lecture seule du fichier donnees1.txt */
    if (fich1 == NULL)
    {
        printf("Erreur à l'ouverture de donnees1.txt\n");
        return 1;
    }

    fich2 = fopen("donnees2.dat", "w+"); /* Ouverture en lecture et écriture du fichier donnees2.txt */
    if (fich2 == NULL)
    {
        printf("Erreur à l'ouverture de donnees2.txt\n");
        return 1;
    }

    fgets(ligne, 50, fich1);
    fputs(ligne, fich2);

    fclose(fich1);
    fich1 = NULL;
    fclose(fich2);
    fich2 = NULL;

    return 0;
}
```

3.3.2 Gestion des tampons

Lors de la lecture d'un caractère, à la validation par la touche <entrée>, le retour à la ligne est conservé dans le tampon du flot, et sera donc attribué à la prochaine variable lue.

Exemple

```
char a = 0, b = 0 ;

scanf("%c",&a) ;       /* si l'utilisateur rentre 'a'<entrée> */
scanf("%c",&b) ;       /* alors b = '\n' */
```

Ce problème est très général et invite à faire très attention à l'état des tampons. Quand nécessaire, la fonction ci dessous (non fournie par défaut) permet de vider le tampon associé à un flot donné.

Exemple

```
void viderbuffer(FILE *stream)
{
    char c = 0 ;
    while (c != '\n' && c != EOF)
    {
        c = fgetc(stream) ;
    }
}
```

3.3.3 Longueur de la chaîne

Si scanf lit une chaîne de caractères plus grande que celle prévue lors de la déclaration, **il y a un risque d'écrasement d'une autre variable du code !**

Exemple

```
int i = 3 ;
char tab[4] = "" ;

scanf("%s", tab) ;              /* si l'utilisateur rentre "un_mot_tres_long" */
printf("tab = %s\n", tab) ;     /* le programme affiche : tab = un_mot_tres_long */
printf("i = %d\n", i) ;         /* le programme affiche : i = 1735290732 */
```

Pour résoudre ce problème, il faut toujours spécifier la longueur de la chaîne à lire dans scanf. Celle-ci doit être inférieure à la longueur de la chaine déclarée. Ceci ne dispense pas de vider les tampons quand nécessaire.

3.3.4 Manipulations classiques

Le C ne sait faire que des affectations et des comparaisons pour 1 seul caractère (sauf au moment de la déclaration).

Exemple

```
char p = 0, q = 0 ;
char ch1[10], ch2[10] = "Hello" ;       /* valide */

p = 'A' ;                               /* valide */
ch1 = "Bonjour" ;                       /* non valide */
if (p == q) ;                           /* valide */
if (ch1 == ch2)                         /* non valide */
```

Pour réaliser ce type d'opération, on peut utiliser les fonctions définies dans <string.h>, en particulier **strncpy** et **strncmp** (voir chapitre précédent).

Exemple

```
#include <stdio.h>
#include <string.h>
int main (void)
{
    char chaine1[10] = "" ;
    char chaine2[10] = "" ;

    strncpy(chaine1, "Bonjour", 10) ;          /* Met "Bonjour" dans chaine1 */
    strncpy(chaine2, "Hello", 10) ;            /* Met "Hello" dans chaine2 */

    if ( strncmp(chaine1,chaine2, 10) == 0)    /* Teste si les n=10 premiers caractères des chaines 1 et 2 sont égaux */
    {• • •}

    strncat(chaine1, chaine2, 10) ;            /* Concatène chaine1 et chaine2 dans la limite de 10 caractères */
                                               /* Chaine1 contient donc "BonjourHe" */

    printf("%d\n", strlen(chaine2)) ;          /* Affiche la longueur effective de chaine2 */

    return 0 ;
}
```

Table des matières

Chapitre 4

Instructions de contrôle

4.1 Rappel sur les instructions et les blocs

- Une expression (e.g. `x = 0`) devient une instruction lorsqu'elle est suivie d'un " ;"
- Les accolades regroupent des instructions en bloc, syntaxiquement équivalent à une instruction unique. L'accolade fermante qui termine un bloc n'est pas suivie de " ;".

4.2 Instructions conditionnelles

4.2.1 Tests `if-else` et `else if`

```
if (expression)
{
   instruction 1 ;
}
else
{
   instruction 2 ;
}
```

- L'expression testée est vraie si la valeur qu'elle renvoie est non nulle.
- La partie `else` est facultative. Quand utilisé dans une séquence de `if`, un `else` s'associe automatiquement au `if` le plus proche d'un même bloc. Pour forcer une association, il faut utiliser des accolades.
- Les instructions `else if` ne sont que des applications du principe précédent.

```
if (expression 1)
{
   instruction 1 ;
}
else if (expression 2)
{
   instruction 2 ;
}
else if (expression 3)
{
   instruction 3 ;
}
```

Exemple

```
#define NCAR 30
int main (void)
{
    char chaine[NCAR] = "" ;

    fgets (chaine, NCAR-1, stdin) ;
    if ( chaine[15] )
        printf ("La chaine lue contient plus de 16 caractères\n") ;
    else
        printf ("La chaine lue contient moins de 16 caractères\n") ;
}
```

Exemple

```
double x = 1, y = 0 ;
double a = 0, b = 2 ;

if ( x > 0 )
    if (a > b)
    {
        y = sqrt(x) + sqrt(a-b) ;
    }
else                                /* Malgré l'indentation, le else est associé au if le plus proche */
    y = sqrt(-x) ;                  /* A la suite de cet exemple, y = nan */
printf ("y = %lf\n", y) ;
```

4.2.2 Branchement `switch`

L'instruction switch effectue des instructions en fonction de la valeur d'un entier.

```
switch(i)
{
case val1 :
   instruction 1 ;
   instruction 2 ;
   break ;
case val2 :
case val3 :
   instruction 3 ;
   break ;
default :
   instruction 4 ;
   break ;
}
```

- L'instruction `break` provoque la sortie immédiate du `switch`.
- Si un cas est trouvé, le programme effectue **toutes les instructions suivantes** (y compris celles des autres cas) jusqu'au prochain `break`.
- Si il n'y a pas le cas `default` et qu'aucun autre cas n'est rencontré le programme plante.

4.2.3 Opérateur conditionnel ? :

Cet opérateur est un test `if` écrit sous forme condensée que l'on retrouve parfois dans des codes C.

```
expr1 = expr2 ? expr3 : expr4
```

est équivalent à

```
if (expr2)
   expr1 = expr3 ;
else
   expr1 = expr4 ;
```

Exemple

```
z = (a > b) ? a : b ;      /* équivalent à z = max(a,b) */
```

4.3 Boucles

4.3.1 Boucle `while`

```
while (expression)
{
   instruction ;
   ...
}
```

permet d'effectuer des instructions tant que l'expression est vrai (i.e. `expression != 0`).

4.3.2 Boucle `do while`

```
do
{
   instruction ;
   ...
} while (expression) ;
```

est équivalente à `while`, sauf que les instructions sont effectuées au moins une fois.

4.3.3 Boucle `for`

```
for (expr1 ; expr2 ; expr3)
{
   instruction ;
   ...
}
```

est équivalent à

```
expr1 ;
while (expr2)
{
   instruction ;
   ...
   expr3 ;
}
```

- L'instruction `for` est bien plus versatile que le `DO` du fortran : la boucle n'est pas nécessairement arithmétique ; si il y a un compteur, celui-ci peut être modifié dans le corps des instructions.
- Attention à l'utilisation des " ;" dans la syntaxe d'un `for`.

Exemple

```
/* exemple de boucle arithmétique */
char chaine[8] = "Bonjour" ;
int i = 0 ;

for ( i = 0 ; i < 8 ; i++ )
{
    printf("%c", chaine[i]) ;
}
printf("\n") ;
```

Exemple

```
/* exercice : y'a-t-il une difference entre les 2 instructions suivantes */

for (i = 0 ; i < N ; i++ )

for (i = 0 ; i < N ; ++i )
```

4.3.4 Instructions `break`, `continue` et `return`

- L'instruction `break` provoque la sortie immédiate de la boucle ou du switch le plus proche.
- L'instruction `continue` relance immédiatement la boucle dans laquelle elle se trouve. Pour une boucle `while`, la condition d'arrêt est réévaluée. Pour une boucle for, on passe à l'instruction finale.
- L'instruction `return` provoque la sortie de la fonction.

4.3.5 Instruction `goto` et étiquettes

Déconseillée, `goto` se retrouve quelquefois dans des programmes afin de sortir de boucles imbriquées.

```
goto etiquette ;
...
etiquette :
```

Table des matières

Chapitre 5

Les tableaux

5.1 Déclaration et utilisation

Les tableaux sont une suite de variables de même type, situées dans un espace contigu en mémoire.

```
type tableau[n] ;
```

- Par définition, `tableau` **est un pointeur** sur la première case (i.e. variable contenant l'adresse de cette case).
- En C89, `n` ne peut pas être une variable, mais uniquement une constante littérale ou définie via `#define`.
- Contrairement au fortran, les indices des tableaux varient de 0 à n-1.
- L'appel à l'élément `i` du tableau se fait via `tableau[i]`.
- La taille d'un tableau n'est pas associée au nom de la variable. Le langage n'interdit donc pas d'appeler un élément situé au delà des limites du tableau.

Exemple

```
/* Faut-il utiliser un & ou non ? */
char c = ' ';
char chaine[8] = "";

scanf("%c", &c);
scanf("%7s", chaine);          /* lecture de la chaine. & n'est pas nécessaire car chaine est une adresse */
scanf("%c", &chaine[3]);       /* remplacer la 4ème lettre de la chaine. & est nécessaire */
```

Exemple

```
#define M 8

int main (void)
{
    const int N = 8;
    int chaine[N];      /* non valide */
    int chaine[M];      /* valide */
    ...
}
```

Exemple

```
int b = 15 ;
int a[4] = {0} ;

printf("%d\n", a[7]) ;       /* Affiche la valeur de l'élément situé à cette adresse, ici la valeur de b */
```

5.2 Initialisation

```
type tableau[n] = { val1, val2, val3, ...} ;
```

- Ce type d'initialisation n'est possible qu'au moment de la déclaration.
- Si un élément est initialisé, tous les éléments non initialisés sont intialisés à 0.
- L'initialisation permet d'ajuster automatiquement la taille d'un tableau.
- Lors de la déclaration, une chaîne de caractères peut aussi être intialisée via `char chaine[n] = "chaine"`

Exemple

```
int tab1[] = { 3, 6, 9 } ;              /* ajuste la taille de tab1 à 3 éléments */
int tab2[5] = { 3, 6, 9 } ;             /* initialise tab2[3] et tab2[4] à 0 */
char chaine[8] = "Bonjour" ;
```

5.3 Tableau à plusieurs dimensions

```
type tableau[n1][n2][n3]... ;
```

- Un tableau à plusieurs dimensions est un tableau uni-dimensionnel de dimension `n1` dont chaque composante est un tableau uni-dimensionnel de dimension `n2` dont chaque élément est un tableau ...
- Lors de l'initialisation, l'indice le plus intérieur varie en premier.
- Les éléments du tableau sont stockés dans un espace contigu en mémoire.

Exemple

```
int tab[2][3] = { { 1, 2, 3 } , { 4, 5, 6 } } ;

tab[0][3] = 10 ;                        /* erreur de programmation : dépassement des bornes */

printf("%d\n", tab[1][0]) ;             /* affiche 10 */
```

Table des matières

Chapitre 6

Fonctions et programmation modulaire

6.1 Définition & fonctionnement

6.1.1 Définition de fonction

La définition d'une fonction est la description complète de ses arguments, variables internes et instructions

```
type_retour nom_fonction (type1 arg1, type2 arg2, type3 arg3, ...)
{
   type_a var_interne_a ;
   type_b var_interne_b ;

   instructions ;

   return expression ;
}
```

- Toute fonction renvoie une seule valeur : `expression`. Le type de cette valeur (entier, flottant, pointeur, ...) est indiqué par `type_retour`. Par défaut, si `type_retour` n'est pas indiqué, toute fonction renvoie un entier.
- Une fonction reçoit des arguments en entrée. Leurs types et noms sont déclarés en première ligne.
- Si une fonction ne renvoie aucune valeur ou ne prend aucun argument, utiliser le mot clef **void**. Dans le cas où il n'y a pas d'argument, l'appel à la fonction nécessite tout de même des parenthèses : `nom_fonction()`.
- L'instruction **return** marque la fin de la fonction. Les instructions suivantes sont ignorées.
- Une fonction peut-être définie n'importe où : avant ou après la fonction `main`, dans le même fichier ou non.

Exemple

```
float absolue (float x)
{
    if (x < 0)
        return -x;
    else
        return x;
}
```

Exemple

```
double pi (void)
{
    return 4*atan(1.0) ;
}
```

Exemple

```
void mess_err (void)
{
    printf ("Une erreur s'est produite\n") ;
    return ;
}
```

6.1.2 Passage des arguments par valeur

- En C, les arguments d'une fonction sont passés par valeur. La fonction appelée travaille sur des copies des variables de la fonction appelante. Celles-ci sont locales à la fonction appelée.
- Pour modifier une variable passée en argument, il faut donner l'adresse mémoire de la variable (pointeur).

Exemple

```
int plus_dix (int) ;

int main (void)
{
    int i = 15, l = 0 ;          /* la variable i n'a rien à voir avec celle de la fonction plus_dix */

    l = plus_dix(i) ;

    printf ("i = %d\n", i) ;     /* affiche i = 15 */
    printf ("l = %d\n", l) ;     /* affiche l = 25 */

    return 0 ;
}

int plus_dix (int i)
{
    i = i + 10 ;
    return i ;
}
```

6.1.3 Les arguments de la fonction `main`

Au lancement d'un programme C, les arguments sur la ligne de commande peuvent être transmis à la fonction `main`.

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

- L'entier `argc` contient alors le nombre d'arguments transmis + 1.
- Les arguments sont des chaînes de caractères (sans espace) stockées dans le tableau à deux dimensions `argv`.

Exemple

```
#include <stdio.h>
int main (int argc, char **argv)
{
    int i = 0 ;

    printf ("Le programme a reçu %d arguments\n", argc) ;

    for (i = 0 ; i < argc ; i++)
        printf ("- arg %d : %s\n", i, argv[i]) ;

    return 0 ;
}
```

Lorsque le programme précédent est lancé avec la ligne de commande

`./mon_prog a 5 toto 8.54678 Oscar`

le tableau `argv` se remplit de la manière suivante

argv[0]	m	o	n	_	p	r	o	g
argv[1]	a							
argv[2]	5							
argv[3]	t	o	t	o				
argv[4]	8	.	5	4	6	7	8	
argv[5]	0	s	c	a	r			

6.2 Portée et visibilité des fonctions

6.2.1 Prototype de fonction

Pour utiliser une fonction, la fonction appelante doit connaître toutes les propriétés de la fonction appelée : son nom, le type de valeur retournée, le nombre et le type de chaque argument. Pour cela, on utilise un **prototype de fonction**.

```
1 type_retour nom_fonction (type1, type2, type3, ...) ;
```

- Le prototype d'une fonction doit être placé avant tout appel à la fonction.
- Le nom des arguments est inutile. Seul leur type importe.
- Ne pas oublier le " ;" qui indique qu'il s'agit d'un prototype et non d'une définition de fonction.

6.2.2 Portée des fonctions

- Un programme peut être composé de nombreux fichiers sources : e.g. main.c, fichier1.c, fichier2.c, ... chacun contenant les définitions de plusieurs fonctions, réunies selon une certaine logique.

- Par défaut, une fonction peut être utilisée par n'importe quel fichier, du moment que son prototype est connu.
- Si l'attribut `static` est utilisé, la fonction ne peut être utilisée que par les fonctions du même fichier.

main.c

```c
#include <stdio.h>
#include "fichier1.h"

typeA foncA (...) ;
typeD foncD (...) ;

int main (void)
{
     ...
     foncA(...) ;
     foncB(...) ;
     foncC(...) ;
     foncD(...) ;
     ...
}

typeA foncA (...)
{...}
```

fichier1.h

```c
typeB foncB (...) ;
typeC foncC (...) ;
```

fichier1.c

```c
typeB foncB (...)
{...}

typeC foncC (...)
{...}
```

fichier2.c

```c
typeD foncD (...)
{...}

static typeE foncE (...)
{...}
```

6.2.3 Header

Il est souvent utile de réunir les prototypes de plusieurs fonctions dans un fichier séparé, dit "header" : par exemple toutes les fonctions de fichier1.c dans fichier1.h (cf ci-dessus). On inclut alors les prototypes où nécessaires via

```
#include "nom_de_fichier.h"
```

6.2.4 Compilation groupée ou séparée

Un programme écrit sur un seul fichier se compile via

```
gcc -ansi -Wall -Wextra fichier1.c -o executable
```

Un programme écrit sur plusieurs fichiers peut se compiler via

```
gcc -ansi -Wall -Wextra fichier1.c fichier2.c ... -o executable
```

Il est toutefois déconseillé de faire cela car cette méthode cache les étapes intermédiaires de la compilation (e.g. suppression des fichiers .o) et conduit à des erreurs de logique de la part du programmeur. A la place, on recommande d'effectuer les étapes suivantes

```
gcc -c -ansi -Wall -Wextra fichier1.c
gcc -c -ansi -Wall -Wextra fichier2.c
...
gcc fichier1.o fichier2.o ... -o executable
```

i.e. de créer d'abord tous les fichiers objets .o (cf Chap. 1) en compilant chaque fichier .c de manière séparée, puis d'effectuer l'étape d'édition de lien en mettant bout à bout tous les fichiers .o nécessaires pour construire l'exécutable.

Garder les fichiers .o permet de ne pas recompiler un fichier qui n'a pas été modifié. Ceci est à la base de la construction de `makefile` qui visent à automatiser la compilation de programmes (voir projet).

Table des matières

Chapitre 7

Pointeurs

7.1 Mémoire et adresses

7.1.1 Stockage des variables en mémoire

À la déclaration, le programme demande à l'OS de réserver des places mémoire pour chaque variable. Cette place est repérée par une **adresse mémoire**. La valeur de la variable est stockée à cette adresse sur le nombre d'octets nécessaire.

```
float f = 3.14 ;
int a = 14 ;
short b = 100 ;
char c[5] = "test" ;
```


nom	adresse (hexadecimal)	valeur (hexadecimal)
c[0]	5073cac0	74
c[1]	5073cac1	65
c[2]	5073cac2	73
c[3]	5073cac3	74
c[4]	5073cac4	00
b	5073cac5	00 64
a	5073cac7	00 00 00 0e
f	5073cacb	00 00 0e 00

7.1.2 Obtenir l'adresse d'une variable : opérateur &

- L'adresse de toute variable peut être obtenue avec l'opérateur unaire `&`.
- Cet opérateur ne s'applique ni aux expressions, ni aux constantes de précompilation, ni aux variables register.

Exemple

```
#define N 100
...
int a = 10 ;
char c[6] = "test" ;
register int b = 0 ;
printf ("adresse c[0] = %x\n", &c[0]) ;      /* valide */
printf ("adresse c[1] = %x\n", &c[1]) ;      /* valide */
printf ("adresse a    = %x\n", &a) ;         /* valide */
printf ("adresse b    = %x\n", &b) ;         /* non valide */
printf ("adresse N    = %x\n", &N) ;         /* non valide */
```

7.2 Manipulations de pointeurs

7.2.1 Déclaration et initialisation

Un pointeur est une variable faite pour contenir une adresse mémoire. On le déclare et initialise via

```
type *nom_pointeur = NULL ;
```

- Un pointeur est associé à un type de variable. Il ne peut contenir que l'adresse d'une variable du même type.
- `NULL` est une constante de la bibliothèque standard représentant une adresse invalide. Initialiser un pointeur à cette valeur signifie que le pointeur ne contient initialement aucune adresse.

7.2.2 Affectation

Pour affecter l'adresse d'une variable `a` au pointeur `p`, on écrit

```
p = &a ;
```

On dit que `p` pointe sur `a`. Au niveau de la mémoire, on obtient le schéma suivant

```
int a = 14 ;
int *p = NULL ;
p = &a ;
```


nom	adresse (hexadecimal)	valeur (hexadecimal)
a	5073cac7	00 00 00 0e
p	5073cacb	50 73 ca c7

7.2.3 Opérateur d'indirection : *

- L'opérateur d'indirection `*` donne accès à la valeur de la variable pointée.
- Si `p` pointe sur `a`, manipuler `*p` revient à manipuler `a`.
- `*` et `&` sont des opérateurs réciproques : `*&a` et `a` sont équivalents.

Exemple

```
int x = 1, y = 2 ;
int z[10] = 0 ;
int *p = NULL ;              /* p est un pointeur sur un entier */

p = &x ;                     /* p pointe maintenant sur x */
y = *p ;                     /* y vaut désormais 1 */
*p = 0 ;                     /* x vaut désormais 0 */
p = &z[1] ;                  /* p pointe maintenant sur z[1] */
*p = 10 ;                    /* z[1] vaut désormais 10 */
z[1] = 11 ;                  /* z[1] vaut désormais 11 */
x = *p ;                     /* x vaut désormais 11 */
p = z ;                      /* p pointe maintenant sur z[0] */
```

7.3 Pointeurs et arguments de fonction

Un atout des pointeurs est de permettre qu'une fonction appelée modifie les variables d'une fonction appelante.

7.3.1 Position du problème

Imaginons que l'on souhaite créer une fonction qui renvoie plusieurs valeurs : par exemple une fonction qui permute un couple de variables. Comme en C les arguments sont passés par valeurs, le programme suivant échoue

Exemple

```
void permute (int, int) ;

int main (void)
{
    int i = 1, j = 2 ;
    permute (i, j) ;
    printf ("i, j = %2d, %2d\n", i, j) ;      /* affiche i, j = 1, 2 */
    return 0 ;
}

void permute (int a, int b)                   /* rappel : les variables a et b sont locales à la fonction */
{
    int temp = a ;
    a = b ;
    b = temp ;
    return ;
}
```

7.3.2 Solution

Pour résoudre le problème, il faut passer en argument de la fonction les adresses des variables à modifier.

Exemple

```
void permute (int *, int *) ;

int main (void)
{
    ...
    permute (&i, &j) ;              /* On transmet à la fonction les adresses en mémoire des variables à modifier */
    ...
}

void permute (int *a, int *b)       /* Les arguments de la fonction sont donc des pointeurs */
{
    int temp = *a ;
    *a = *b ;
    *b = temp ;
    return ;
}
```

Lors de l'appel à la fonction, les variables locales de type pointeur `a` et `b` reçoivent les adresses des variables `i` et `j` de la fonction appelante. L'opérateur `*` permet alors de modifier les valeurs des variables stockées à ces adresses.

7.4 Pointeurs et tableaux

En C, les pointeurs et les tableaux sont fortement liés.

- Par définition, le **nom d'un tableau est un pointeur** sur la première case du tableau : `tab` ⇔ `&tab[0]`
- Si `p` pointe sur un élément d'un tableau, `p-1` pointe sur l'élément précédent, `p+1` sur l'élément suivant et `p+n` sur le n-ième élément suivant. L'adresse change du nombre d'octets nécessaire à chaque élément.
- On peut appliquer un indice à tout pointeur : `p[i]` est la valeur de la variable situé à l'adresse `p+i`, i.e. `*(p+i)`.

7.5 Pointeurs de fonctions

- En C, une fonction n'est pas une variable mais il est possible de définir des pointeurs de fonction que l'on peut affecter, placer dans des tableaux, passer en arguments à d'autres fonctions, et ainsi de suite.
- Un avantage majeur est de pouvoir créer une fonction `f` qui utilise comme argument une autre fonction, par exemple `g` ou `h`. Pour cela il est nécessaire que `g` et `h` aient le même prototype.
- Par définition, le nom de toute fonction est un pointeur sur la première instruction de la fonction. Une fonction composée (comme évoquée ci-dessus) s'écrit alors `f(h)` et non pas `f(&h)` (comme pour un tableau).

7.5.1 Déclaration

Un pointeur de fonction se déclare ainsi

```
typeF (*pf) (type1, type2, type3, ...) ;
```

Cette instruction indique que la fonction pointée renvoie une valeur de type `typeF` et prend comme arguments plusieurs variables de types `type1`, `type2`, `type3`...

Exemple

```
/* Ne pas confondre les deux déclaration suivantes */

int (*pf) (float, float) ;      /* Pointeur sur une fonction qui renvoie un entier et a pour arguments 2 réels */

int *pf (float, float) ;        /* Fonction qui renvoie un pointeur sur une entier et a pour arguments 2 réels */
```

7.5.2 Exemple d'utilisation

Imaginons qu'on veuille construire une fonction `integrale` qui calcule l'intégrale de n'importe quelle fonction entre deux bornes `x` et `y` fixées par l'utilisateur, en utilisant la méthode des trapèzes.

Exemple

```
#include <stdio.h>
#define N 100

double integrale (double, double, double (*) (double) ) ;
double triple (double) ;
double quadruple (double) ;

int main (void)
{
    double a = 0.0 ;
    double b = 3.0 ;
    double integ1 = 0.0 ;
    double integ2 = 0.0 ;

    integ1 = integrale (a, b, triple ) ;          /* Calcul de l'intégrale de triple. triple est un pointeur */
    integ2 = integrale (a, b, quadruple ) ;       /* Calcul de l'intégrale de quadruple. quadruple est un pointeur */
    printf("integ1 = %lf\n", integ1) ;            /* Affichage de la première intégrale = 13.5 */
    printf("integ2 = %lf\n", integ2) ;            /* Affichage de la seconde intégrale = 18 */

    return 0 ;
}

double integrale (double x, double y, double (*f)(double) )
{
    double z = 0.0 ;
    double integ = 0.0 ;
    double dz = 0.0 ;
    int i = 0 ;

    dz = (y-x) / N ;
    integ = f(x) * dz / 2 + f(y) * dz / 2 ;
    for (i = 1 ; i < N ; i++)
    {
        z = x + i * dz ;
        integ = integ + f(z) * dz ;
    }

    return integ ;
}

double triple (double x)
{
    return 3*x ;
}

double quadruple (double x)
{
    return 4*x ;
}
```

Table des matières

Chapitre 8

Allocation dynamique

8.1 Préliminaires

8.1.1 Limites de l'allocation automatique

Par défaut, l'espace mémoire nécessaire pour stocker des variables est géré automatiquement par le programme. Dans chaque fonction, la mémoire requise pour stocker une variable est allouée lors de la déclaration et libérée lors de la sortie de la fonction. Cette approche pose toutefois des problèmes, notamment lors de la manipulation des tableaux.

- L'espace mémoire disponible via l'allocation automatique est limité. Cette limite dépend de la machine.
- Si la taille du tableau n'est pas connue a priori, cette approche force à déclarer un tableau de grande taille quitte à n'en utiliser qu'une partie lors de l'exécution, i.e. force à réserver inutilement de l'espace mémoire.
- Créer un tableau multidimensionnel avec un nombre variable de colonnes par ligne n'est pas possible.
- Le passage d'un tableau à plusieurs dimensions en argument de fonction est difficile.

Pour résoudre ces problèmes, il faut faire appel à **l'allocation dynamique**. L'idée est de **gérer soi-même la mémoire**.

8.1.2 Taille d'une variable ou d'un type : opérateur `sizeof`

L'opérateur unaire `sizeof` permet d'obtenir la taille d'un type, d'une variable, ou d'une constante. Cette taille est exprimée en octets sur un entier non signé de type `size_t` qui dépend de la machine (e.g. `long`).

Exemple

```
#include <stdio.h>

int main (void)
{
    char c = 'e' ;
    int i = 1 ;
    double f[3] = {1,2,3} ;

    printf("size = %lu\n", sizeof (char)) ;              /* affiche size = 1 */
    printf("size = %lu\n", sizeof (double)) ;            /* affiche size = 8 */
    printf("size = %lu\n", sizeof (c)) ;                 /* affiche size = 1 */
    printf("size = %lu\n", sizeof (i)) ;                 /* affiche size = 4 */
    printf("size = %lu\n", sizeof (f)) ;                 /* affiche size = 24 */
    printf("size = %lu\n", sizeof ("Bonjour !")) ;       /* affiche size = 10 */

    return 0 ;
}
```

8.2 Allocation dynamique

8.2.1 Méthode

Pour réaliser une allocation dynamique de mémoire, il faut systématiquement suivre trois étapes.

1. Demander de la mémoire au système d'exploitation, e.g. via la fonction `malloc`.

2. Vérifier que la mémoire a bien été allouée par le système d'exploitation.

3. Libérer la mémoire après utilisation, e.g. via la fonction `free`. Si cette étape n'est pas faîte proprement, le programme s'expose à des fuites de mémoire, i.e. risque d'utiliser plus de mémoire que nécessaire.

8.2.2 Allocation : fonctions `malloc` et `calloc`

```
#include <stdlib.h>
...
type *nom_pointeur = NULL ;
int  nombre = ... ;
...
nom_pointeur = malloc ( nombre * sizeof(type) ) ;
```

- La fonction `malloc` fait partie de la bibliothèque `<stdlib.h>`. Son prototype est

  `void *malloc (size_t) ;`

- La fonction prend comme argument le nombre d'octets à réserver, codé sur un entier de type `size_t`. Il est donc courant d'utiliser le résultat de l'opérateur `sizeof` comme argument.

- Elle renvoie un pointeur contenant l'adresse du bloc alloué. Comme cette fonction permet de réserver un espace mémoire contenant n'importe quel type de variables, l'adresse renvoyée est **non typée** : `void *`. Ci-dessus, il y a donc une conversion implicite quand `nom_pointeur` est affecté au résultat de `malloc`.

  `nom_pointeur = malloc (...) ;` ⇔ `nom_pointeur = (type *) malloc (...) ;`

- Si une erreur se produit (e.g. plus de mémoire disponible), la fonction renvoie `NULL`.

Exemple

```
int *pi = NULL ;
double *pf = NULL ;
int *pp = NULL ;
int n = 20 ;

pi = malloc (4) ;                  /* fournit l'adresse d'un bloc de 4 octets et l'affecte à pi */
pf = malloc ( sizeof (double) ) ;  /* fournit l'adresse d'un bloc de 8 octets et l'affecte à pf */
pp = malloc ( n * sizeof (int) ) ; /* fournit l'adresse d'un bloc de 80 octets et l'affecte à pp */
```

Une autre fonction est souvent utilisée pour allouer de la mémoire en C : `calloc`. Son prototype est `void *calloc (size_t, size_t) ;`. Similaire à `malloc`, `calloc` alloue plusieurs blocs de même taille et met à 0 les bits alloués.

```
nom_pointeur = calloc (nombre, sizeof(type)) ;
```

8.2.3 Test de l'allocation

Toute allocation de mémoire doit être testée. En cas d'échec, le code doit être arrêté. Il existe plusieurs méthodes

```
#include <stdlib.h>
...
nom_pointeur = malloc (nombre_octets) ;
if (nom_pointeur == NULL)
   exit(-1) ;
```

```
#include <stdlib.h>
#include <err.h>
...
nom_pointeur = malloc (nombre_octets) ;
if (nom_pointeur == NULL)
   err(-1, "%s:%d malloc (%lu) failed", __FILE__, __LINE__, nombre_octets) ;
```

- La fonction `exit` fait partie de la bibliothèque `<stdlib.h>`. Elle arrête le programme immédiatement et prend en argument la valeur que le programme doit retourner.
- La fonction `err` est dans la bibliothèque `<err.h>` (non standard mais généralement installée avec gcc). Elle arrête le programme et prend en argument la valeur que le programme doit retourner et un message à afficher.
- `__FILE__` et `__LINE__` sont des macros prédéfinies. `__FILE__` donne le nom du fichier courant sous la forme d'une chaine de caractères. `__LINE__` donne la ligne courante.

Exemple

```
int *p = NULL;
int n = -1000;

p = malloc ( n * sizeof(int) );
if ( p == NULL )
    err(-1, "%s :%d malloc (%lu) failed", __FILE__, __LINE__, n * sizeof(int) );
```

8.2.4 Libérer la mémoire : fonction `free`

```
#include <stdlib.h>
...
free (nom_pointeur) ;
nom_pointeur = NULL ;
```

- La fonction `free` fait partie de la bibliothèque `<stdlib.h>`. Son prototype est `void free (void *)`. Elle prend comme argument l'adresse du bloc mémoire à libérer, quel que soit le type de variables stockées.
- Un bloc libre ne peut être désalloué. Il est capital de forcer le pointeur que l'on vient de libérer à la valeur NULL.

8.3 Exemples d'allocations dynamiques

8.3.1 Tableau à une dimension

Exemple

```
/* Réserver de la place pour un tableau de n entiers, où n est lu au clavier */
/* Le tableau est rempli avec le carré des indices */

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

int main (void)
{
    int *tab = NULL ;
    int i = 0, n = 0 ;

    printf("taille du tableau ? n = ") ;
    scanf("%d", &n) ;

    tab = malloc ( n * sizeof (int) ) ;
    if ( tab == NULL )
        err(-1, "%s :%d malloc (%lu) failed", __FILE__, __LINE__, n * sizeof (int) ) ;

    for (i = 0 ; i < n ; i++)
        tab[i] = i * i ;

    free (tab) ;
    tab = NULL ;

    return 0 ;
}
```

- Créer un tableau revient à réserver un bloc pouvant stocker `n` éléments du même type, e.g. `n * sizeof (int)`.
- L'adresse de ce bloc mémoire est stockée dans un pointeur, e.g. `tab`. Les valeurs des éléments successifs sont alors accessibles en appliquant un indice au pointeur : `tab[0]`, `tab[1]`, ..., `tab[i]`, ...

Mémoire avant allocation

nom	adresse (hexadecimal)	valeur (hexadecimal)
`n`	`52a67aac`	00 00 00 04
`tab`	`52a67ab0`	00 00 00 00

Mémoire après allocation et affectation

nom	adresse (hexadecimal)	valeur (hexadecimal)
`n`	`52a67aac`	00 00 00 04
`tab`	`52a67ab0`	da 50 00 00
`tab[0]`	`da500000`	00 00 00 00
`tab[1]`	`da500004`	00 00 00 01
`tab[2]`	`da500008`	00 00 00 04
`tab[3]`	`da50000c`	00 00 00 09

8.3.2 Tableau à plusieurs dimensions

Exemple

```
/* Réserver de la place pour un tableau de n*p entiers, où n et p sont lus au clavier */

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

int main (void)
{
    int **tab = NULL ;
    int i = 0, j = 0, p = 0, n = 0 ;

    printf("taille du tableau ? n p = ") ;
    scanf("%d%d", &n, &p) ;

    tab = malloc ( n * sizeof (int *) ) ;
    if ( tab == NULL )
        err(-1, "%s :%d malloc (%lu) failed", __FILE__, __LINE__, n * sizeof (int *) ) ;
    for (i = 0 ; i < n ; i++)
    {
        tab[i] = malloc ( p * sizeof (int) ) ;
        if ( tab[i] == NULL )
            err(-1, "%s :%d malloc (%lu) failed", __FILE__, __LINE__, p * sizeof (int) ) ;
    }

    for (i = 0 ; i < n ; i++)
        for (j = 0 ; j < p ; j++)
            tab[i][j] = p * i + j + 1 ;

    for ( i = 0 ; i < n ; i++)
    {
        free ( tab[i] ) ;
        tab[i] = NULL ;
    }
    free (tab) ;
    tab = NULL ;
    return 0 ;
}
```

- Créer un tableau à deux dimensions passe par plusieurs étapes.
  1. Réserver un bloc mémoire pouvant stocker les adresses de chaque sous tableau, e.g. `n * sizeof (int *)`
  2. Stocker l'adresse de ce bloc mémoire dans un pointeur de pointeur, e.g. `tab`.
  3. Réserver des blocs pour stocker les éléments de chaque sous tableau, e.g. `p * sizeof (int)`
  4. Stocker les adresses de ces blocs dans le tableau de pointeur préalablement défini : `tab[0]`, `tab[1]`, ... Les éléments sont alors accessibles en appliquant un indice à ces pointeurs : `tab[0][0]`, `tab[0][1]`, ...
- Comme la mémoire de chaque sous tableau est réservée séparément, les blocs ne sont pas toujours contigus.
- À la libération, **tous les blocs mémoire doivent être libérés** en commençant par ceux les plus internes.
- La méthode est généralisable à plusieurs dimensions. Il suffit de définir un pointeur de pointeur de pointeur ...

Mémoire avant allocation

nom	adresse (hexadecimal)	valeur (hexadecimal)
n	5c666aa8	00 00 00 03
p	5c666aac	00 00 00 02
tab	5c666ab0	00 00 00 00

Mémoire après allocation et affectation

nom	adresse (hexadecimal)	valeur (hexadecimal)
n	5c666aa8	00 00 00 03
p	5c666aac	00 00 00 02
tab	5c666ab0	da 40 3a 20
tab[0]	da403a20	da 40 3a 40
tab[1]	da403a28	da 40 3a 50
tab[2]	da403a30	da 40 3a 60
tab[0][0]	da403a40	00 00 00 01
tab[0][1]	da403a44	00 00 00 02
tab[1][0]	da403a50	00 00 00 03
tab[1][1]	da403a54	00 00 00 04
tab[2][0]	da403a60	00 00 00 05
tab[2][1]	da403a64	00 00 00 06

8.3.3 Différentes méthodes d'allocation

Il existe différentes méthodes pour allouer la mémoire nécessaire pour stocker des tableaux à plusieurs dimensions. Toutes ont leurs avantages et inconvénients et doivent donc être utilisées en fonction des situations

MÉTHODE	DÉCLARATION	TAILLE	MÉMOIRE
automatique	`int tab[2][3]`	fixée et limitée	contiguë
semi-dynamique	`int *tab[2]`	semi fixée, semi limitée	non contiguë
dynamique (ex 8.3.2)	`int **tab`	non fixée, non limtée	non contiguë
dynamique (ex 8.3.3)	`int **tab`	non fixée, non limtée	contiguë

Exemple

```
/* Réserver de la place pour un tableau de n*p entiers stockés de manière contiguë */
/* Les en-tête à inclure et le début et la fin du programme sont les mêmes que ceux de l'exemple précédent */

int main (void)
{
    ...

    tab = malloc ( n * sizeof (int *) ) ;
    if ( tab == NULL )
        err(-1, "%s :%d malloc (%lu) failed", __FILE__, __LINE__, n * sizeof (int *) ) ;
    tab[0] = malloc ( n * p * sizeof (int) ) ;
    if (tab[0] == NULL)
        err(-1, "%s :%d malloc(%lu) failed", __FILE__, __LINE__, n * p * sizeof (int) ) ;
    for (i = 1 ; i < n ; i++)
        tab[i] = tab[i-1] + p ;       /* Attention, tab[i] est un pointeur : tab[i] + p incrémente l'adresse du pointeur */

    ...
}
```

Table des matières

Chapitre 9

Structures

9.1 Manipulation de structures

En programmation, il est souvent nécessaire de grouper des variables de types différents au sein d'une même entité, dite *enregistrement*. En C, ces entités s'appellent des structures et permettent de créer des variables personnalisées.

9.1.1 Déclaration d'une structure

Déclarer une structure revient à définir un nouveau type de variable. Ceci se fait de la manière suivante

```
struct etiquette
{
   type_a vara ;
   type_b varb ;
   ...
} ;
```

- Le nom donnée à la structure est appelé **étiquette de structure**.
- Une structure peut contenir tout type de variable (entiers, caractères, réels, tableaux, pointeurs, ...). Chaque élément de la structure est appelé **champ**.
- Toute déclaration de structure se termine par un " ;"
- La déclaration s'effectue en début de fichier ou dans un en-tête ".h".

Exemple

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

struct Cercle
{
    struct Point c;
    double r;
};
```

9.1.2 initialisation & manipulation d'une variable structurée

Une variable structurée (de type `struct etiquette`) est déclarée comme une variable normale et s'initialise comme un tableau, champ par champ. Les champs d'une variable structurée sont stockés continûment en mémoire.

```
struct etiquette var_structuree = {...} ;
```

L'accès à un champ de la variable structurée se fait via l'opérateur "."

```
var_structuree.nom_champ
```

Les champs se manipulent comme tout autre variable, comme le montre l'exemple suivant

Exemple

```
/* Exemple d'utilisation des structures déclarées dans l'exemple précédent */
#include <stdio.h>

/* Déclaration des structures */
...

/* Manipulation des structures */
int main(void)
{
    struct Cercle moncercle = {{0,0}, 0} ;

    printf("Centre : x, y ? ") ;
    scanf("%lf%lf", &(moncercle.c.x), &(moncercle.c.y)) ;
    printf("Radius ? ") ;
    scanf("%lf", &(moncercle.r)) ;

    printf("Votre cercle a les propriétés suivantes\n") ;
    printf("- centre : %le, %le\n", moncercle.c.x, moncercle.c.y) ;
    printf("- radius : %le\n", moncercle.r) ;

    return 0 ;
}
```

9.1.3 Alias de structure

Pour simplifier la déclaration de variables structurées, il est possible de définir des alias de type via

```
typedef struct etiquette alias ;
```

Le mot "`alias`" est alors synonyme de "`struct etiquette`". Ces alias doivent être définis en début de fichier. Une variable structurée peut alors être déclarée simplement via

```
alias var_structuree = {...} ;
```

Exemple

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

#define NCAR 100

struct Domicile
{
    int numero ;
    char rue[NCAR] ;
} ;

typedef struct Personne Personne ;
struct Personne
{
    char nom[NCAR] ;
    char prenom[NCAR] ;
    struct Domicile adresse ;
} ;

int main(void)
{
    Personne utilisateur = {"", "", 0, {0,""}} ;
    char *c = NULL ;

    printf("Quel est votre nom ? ") ;
    fgets(utilisateur.nom, NCAR-1, stdin) ;
    if( (c = strchr(utilisateur.nom,'\n')) != NULL ) *c = '\0' ;

    printf("Quel est votre prénom ? ") ;
    fgets(utilisateur.nom, NCAR-1, stdin) ;
    if( (c = strchr(utilisateur.prenom,'\n')) != NULL ) *c = '\0' ;

    printf("Quelle est votre adresse : \n) ;
    printf("- numéro ? ") ;
    scanf("%d", &utilisateur.adresse.numero) ;
    viderbuffer(stdin) ;

    printf("- rue ? ") ;
    fgets(utilisateur.adresse.rue, NCAR-1, stdin) ;
    if( (c = strchr(utilisateur.adresse.rue,'\n')) != NULL ) *c = '\0' ;

    printf ("Vous vous appelez %s %s\n", utilisateur.prenom, utilisateur.nom ) ;
    printf ("\nVous habitez au %d %s\n", utilisateur.adresse.numero, utilisateur.adresse.rue) ;

    return 0 ;
}
```

9.1.4 Pointeur de structure

Comme pour n'importe quelle autre type de variable, il est enfin possible de définir un pointeur de structure.

```
1 struct etiquette *nom_pointeur ;
```

Accéder au champ d'une variable structurée stockée à l'adresse `nom_pointeur` se fait de deux manières différentes

```
(*nom_pointeur).nom_champ
ou
nom_pointeur->nom_champ
```

- Ces deux écritures sont complètement équivalentes.
- Les parenthèses dans la première écriture sont cruciales puisque l'opérateur "." a la priorité sur l'opérateur "*"
- L'opérateur "->" ne s'applique qu'aux pointeurs de structures et pas aux structures.
- En C, il est beaucoup plus fréquent de rencontrer des pointeurs de structures que des structures elles-mêmes. Savoir manier ces deux écritures est donc capital.

9.2 Structures autoréférentielles

9.2.1 Limites des tableaux

Un tableau est une entité conçue pour contenir une liste de valeurs. Que l'espace qu'ils occupe soit alloué automatiquement ou dynamiquement, la taille du tableau doit être connue avant l'allocation. Bien que cette approche comporte de nombreux avantages (accès rapide aux éléments du tableau, simplicité d'écriture), elle est toutefois limitée.

- Pour supprimer ou ajouter un élément à un tableau, il faut créer un nouveau tableau, copier les valeurs à l'intérieur et supprimer l'ancien, ce qui revient à gaspiller de la mémoire et du temps de calcul.
- Intervertir des éléments (ou groupes d'éléments) d'un tableau peut demander de nombreuses manipulations.
- Si aucun espace mémoire contigu n'est suffisant pour contenir un tableau, celui-ci ne sera pas construit.

9.2.2 Concept de liste chaînée

Tous ces problèmes peuvent être résolus avec une **structure autoréférentielle**, i.e. une structure dont l'un des champs est un pointeur sur une structure du même type. Stocker une liste de valeurs consiste alors à

1. allouer la mémoire nécessaire pour stocker une 1ère structure qui contient la 1ère valeur ainsi qu'un pointeur vers la structure suivante ;
2. allouer la mémoire nécessaire pour stocker une 2ème structure qui contient la 2ème valeur ainsi qu'un pointeur vers la structure suivante ;
3. affecter l'adresse de la seconde structure au pointeur de la 1ère structure ;
4. reprendre les étapes 2 et 3 pour les valeurs suivantes.