

UE MP050

Méthodes Numériques et Informatiques - A

Langage C

Albert Hertzog

Jacques.Lefrere@aero.jussieu.fr

Sofian.Teber@lpthe.jussieu.fr

2011–2012

Langage C

2 Types des variables	34
2.1 Types de base	34
2.2 Déclaration et affectation des variables	35
2.2.1 Syntaxe et exemples	36
2.2.2 Valeurs	37
2.3 Domaine et représentation machine des entiers	38
2.3.1 Domaine des entiers non-signés et signés	38
2.3.2 Représentation machine des entiers	40
2.3.3 Structure simplifiée de la RAM : déclaration de deux entiers	41
2.3.4 Structure simplifiée de la RAM : affectation de deux entiers	43
2.4 Valeurs maximales des entiers en C	44
2.4.1 Valeurs maximales des entiers en C : machine 32 bits	45
2.4.2 Valeurs maximales des entiers en C : machine 64 bits	46

Table des matières

1 Introduction	20
1.1 Programmation en langage compilé	20
1.2 Historique du C	22
1.3 Intérêts du C	23
1.4 Généralités	24
1.5 Exemple de programme C avec une seule fonction	25
1.6 Structure générale d'un programme C	26
1.7 Exemple de programme C avec deux fonctions	28
1.8 Compilation	30
1.9 Compilateur	33
2.5 Domaine et précision des flottants	47
2.5.1 Codage des flottants	47
2.5.2 Domaine et précision des flottants	48
2.6 Constantes	49
2.6.1 Syntaxe	49
2.6.2 Exemples d'attribut <code>const</code> en C	50
2.6.3 Exemples d'utilisation de <code>#define</code>	50
3 Opérateurs	52
3.1 Opérateur d'affectation	52
3.2 Opérateurs algébriques	54
3.3 Opérateurs de comparaison	55
3.4 Opérateurs logiques	55

MNI

1

2011-2012

Langage C

3.5	Incrémentation et décrémentation en C	56
3.6	Opérateurs d'affectation composée en C	57
3.7	Opérateur d'alternative en C	57
3.8	Opérateurs agissant sur les bits	58
3.9	Opérateur sizeof en C	59
3.10	Opérateur séquentiel, en C	59
3.11	Opérateurs & et * en C	59
3.12	Priorités des opérateurs en C	60
4	Entrées et sorties standard élémentaires	61
4.1	Généralités	61
4.2	Formats d'affichage en C	63
4.3	Gabarits d'affichage en C	65

4.3.1	Cas des entiers	65
4.3.2	Cas des flottants	65
5	Structures de contrôle	66
5.1	Structure <code>if</code>	67
5.1.1	Exemple de <code>if</code>	71
5.2	Structure <code>switch</code>	73
5.2.1	Exemples de <code>switch-case</code>	74
5.3	Structures itératives ou boucles	78
5.3.1	Boucle définie (<code>for</code>)	78
5.3.2	Exemple de boucle <code>for</code>	80
5.3.3	Boucle indéfinie (<code>while</code> et <code>do...while</code>)	81
5.3.4	Exemple de boucle <code>while</code>	83

5.4	Branchements	84
5.4.1	Exemple de <code>continue</code>	85
5.4.2	Exemple de <code>break</code>	86
6	Introduction aux pointeurs	87
6.1	Intérêt des pointeurs	87
6.2	Déclaration et affectation	88
6.2.1	Déclaration d'une variable ordinaire (rappel)	88
6.2.2	Affectation d'une variable ordinaire (rappel)	89
6.2.3	Déclaration d'un pointeur	90
6.2.4	Affectation d'un pointeur	93
6.3	Indirection (opérateur <code>*</code>)	98
6.4	Initialisation des pointeurs et dissociation	101

6.5	Bilan concernant la syntaxe des pointeurs	103
6.6	Tailles des types de base et adresses en C	104
7	Fonctions en C	106
7.1	Généralités	106
7.2	Définition d'une fonction	107
7.2.1	Exemples de fonctions renvoyant une valeur	109
7.2.2	Exemple d'une fonction sans retour	111
7.2.3	Exemple d'une fonction sans argument	112
7.3	Appel d'une fonction	113
7.3.1	Appel d'une fonction avec retour	113
7.3.2	Appel d'une fonction sans retour	113
7.3.3	Appel d'une fonction sans argument	114

7.4	Déclaration d'une fonction	115
7.4.1	Déclaration au moyen d'un prototype (méthode conseillée)	116
7.4.2	Exemple de déclaration et d'appel d'une fonction	117
7.4.3	Déclaration au moyen de la définition (méthode déconseillée)	118
7.5	Quelques fonctions (standards) du C	119
7.5.1	La fonction principale : <code>main</code>	119
7.5.2	La fonction <code>exit</code>	120
7.5.3	Les fonctions <code>printf</code> et <code>scanf</code>	121
7.5.4	Les fonctions mathématiques (<code>pow</code> , <code>fabs</code> , ...)	122
7.5.5	Exemple d'un programme utilisant <code>math.h</code>	124
7.5.6	Liste des fonctions mathématiques standards	126
7.6	La portée des variables	128
7.6.1	Variables globales	129

8.1	Définition et usage	148
8.1.1	Exemples de programmes élémentaires utilisant des tableaux 1D	149
8.1.2	Exemples de programmes élémentaires utilisant des tableaux 2D	153
8.2	Tableaux de taille fixe	157
8.2.1	Déclaration d'un tableau de taille fixe	157
8.2.2	Indexation et référence à un élément d'un tableau	159
8.2.3	Déclaration d'un tableau 1D : visualisation de la RAM	161
8.2.4	Affectation d'un tableau	162
8.2.5	Affectation d'un tableau 1D : visualisation de la RAM	164
8.2.6	Affectation d'un tableau 2D : visualisation de la RAM	165
8.2.7	Ordre des éléments de tableaux 2D en C	166
8.3	Inconvénients des tableaux de taille fixe	167
8.3.1	Directive préprocesseur (ancienne méthode)	168

7.6.2	Variables locales	129
7.6.3	Exemple d'un programme utilisant une variable globale	130
7.7	Passage de paramètres dans une fonction	132
7.7.1	Exemple de passage par valeur : le faux échange	133
7.7.2	Le faux échange : visualisation de la RAM à l'exécution	136
7.7.3	Exemple de passage par adresse : le vrai échange	139
7.7.4	Le vrai échange : visualisation de la RAM à l'exécution	142
7.7.5	Bilan sur le passage de paramètres	145
7.8	Retour sur <code>printf/scanf</code>	146
7.9	Complément sur les fonctions : la récursivité	147
7.9.1	Exemple de fonction récursive : factorielle	147

8 Tableaux

148

8.3.2	Déclaration tardive et tableau automatique (méthode conseillée)	171
8.3.3	Exemple de programme utilisant un tableau automatique	172
8.4	Tableaux et pointeurs en C	174
8.4.1	Notions de base	174
8.4.2	Arithmétique des pointeurs	175
8.4.3	Retour sur l'ordre des éléments de tableaux 2D en C	176
8.4.4	Sous-tableau 1D avec un pointeur	179
8.4.5	Tableaux 2D et pointeurs	182
8.5	Fonctions et tableaux	185
8.5.1	Passage de tableaux de taille fixe (pour le compilateur)	185
8.5.2	Exemple de passage d'un tableau 1D de taille fixe	186
8.5.3	Exemple de passage d'un tableau 2D de taille fixe	188
8.5.4	Passage de tableau de taille inconnue à la compilation	190

8.5.5	Exemple de passage d'un tableau 1D de taille variable	191
8.5.6	Exemple de passage d'un tableau 2D de taille variable	193
8.5.7	Limite des tableaux automatiques	195
9	Allocation dynamique (sur le tas)	200
9.1	Motivation	200
9.2	Allocation dynamique avec <code>malloc</code> ou <code>calloc</code>	201
9.3	Libération de la mémoire allouée avec <code>free</code>	203
9.4	Attention aux fuites de mémoire	204
9.5	Exemple d'allocation dynamique d'un tableau 1D	205
9.6	Exemple d'allocation dynamique d'un tableau 2D	207
9.7	La bibliothèque <code>libmnitab</code>	211
9.7.1	Exemple de passage d'un tableau dynamique 1D	212

10.5.3	Copie d'une chaîne avec <code>strcpy</code>	229
10.5.4	Comparaison de chaînes avec <code>strcmp</code>	230
10.5.5	Recherche d'un caractère dans une chaîne avec <code> strchr</code>	231
10.5.6	Recherche d'une sous-chaîne dans une chaîne avec <code> strstr</code>	232
11	Entrées–sorties	233
11.1	Introduction	233
11.1.1	Rappel : les fonctions <code>printf</code> et <code>scanf</code>	234
11.1.2	Exemple introductif	235
11.2	Type de fichiers et accès	237
11.3	Ouverture et fermeture d'un fichier	239
11.3.1	Déclaration d'un flux	239
11.3.2	Ouverture d'un flux avec <code>fopen</code>	240

9.7.2	Exemple de passage d'un tableau dynamique 2D	214
9.8	Bilan sur l'allocation de mémoire	216
10	Chaînes de caractères	217
10.1	Définition et usage	217
10.2	Exemple de tableaux de caractères de taille fixe	218
10.3	Exemple de tableaux de caractères de taille quelconque	221
10.4	Déclaration et affectation des chaînes de caractères	224
10.4.1	Chaîne de longueur fixe	224
10.4.2	Chaîne de longueur calculée à l'initialisation	224
10.5	Manipulation des chaînes de caractères	225
10.5.1	Longueur d'une chaîne avec <code>strlen</code>	225
10.5.2	Concaténation de chaînes avec <code>strcat</code>	227

11.3.3	Fermeture d'un flux avec <code>fclose</code>	241
11.3.4	Exemple d'ouverture/fermeture d'un fichier	241
11.4	Entrée-sorties formatées	242
11.4.1	Écriture avec <code>fprintf</code>	242
11.4.2	Lecture avec <code>fscanf</code>	243
11.4.3	Bilan sur les entrées-sorties formatées	244
11.5	Entrées-sorties non formatées (binaires)	245
11.5.1	Lecture avec <code>fread</code>	245
11.5.2	Écriture avec <code>fwrite</code>	246
11.6	Retour sur les formats d'entrée–sortie	247
11.7	Exemple de lecture de fichier formaté en C	249
11.8	Fonctions supplémentaires	253

12 Structures ou types dérivés	255
12.1 Intérêt des structures	255
12.1.1 Exemple introductif de structures	256
12.2 Définition, déclaration et initialisation des structures	259
12.2.1 Définition d'un type structure <code>point</code>	259
12.2.2 Quels types de champs peut-on mettre dans une structure ?	259
12.2.3 Où placer la définition d'une structure ?	260
12.2.4 Déclaration de variables de type structure <code>point</code>	261
12.2.5 Affectation d'une structure (constructeur)	261
12.3 Manipulation des structures	262
12.3.1 Accès aux champs d'une structure	262
12.3.2 Affectation globale (même type)	263
12.3.3 Entrées/sorties et structures	263

12.8 Valeur de retour	277
12.9 Exemple final	278
12.10 Bilan sur les structures	281
13 Éléments de compilation séparée	282
13.1 Introduction	282
13.2 Fichiers d'entête (<i>header files</i>)	284
13.2.1 Définition et usage	284
13.2.2 Structure d'un fichier d'entête	286
13.3 Exemple de programme en plusieurs fichiers	287
13.4 Bibliothèques statiques de fichiers objets	290
13.4.1 Création et utilisation d'une bibliothèque statique (archive)	290
13.4.2 Retour sur la bibliothèque standard	293

12.3.4 Retour sur l'exemple introductif	264
12.4 Représentation en mémoire des structures	267
12.5 Pointeur sur une structure	268
12.6 Exemples de structures (plus ou moins) complexes	269
12.6.1 Tableaux de structures	269
12.6.2 Structures contenant un tableau (taille fixe)	269
12.6.3 Structures contenant un tableau (taille variable)	270
12.6.4 Structures contenant une structure	272
12.6.5 Listes chaînées	273
12.7 Structure et fonction, opérateur flèche	274
12.7.1 Passage par copie de valeur	274
12.7.2 Passage par copie d'adresse	275
12.7.3 Opérateur flèche	276

13.4.3 Retour sur la bibliothèque <code>libm</code>	294
13.4.4 Bilan sur la création et l'usage d'une bibliothèque	295
13.5 Génération d'un fichier exécutable avec <code>make</code>	296
13.5.1 Principe	296
13.5.2 Construction d'un <code>makefile</code>	297
13.5.3 Exemple élémentaire de <code>makefile</code> en C	298
13.5.4 Utilisation d'un <code>makefile</code>	300

14 Conclusions	301
-----------------------	------------

1 Introduction

1.1 Programmation en langage compilé

Conception, écriture et exécution d'instructions destinées à être traitées de manière automatique par un appareil informatique :

- **conception** : définir l'objectif du programme et la méthode à utiliser
⇒ algorithmique
- **codage** : écrire le programme suivant la syntaxe d'un langage de haut niveau, portable et utilisant des bibliothèques : C, fortran, ...
⇒ code source : fichier texte avec instructions commentées compréhensibles pour le concepteur... et les autres
- **compilation** : transformer le code source en un code machine
⇒ code objet puis code exécutable : fichiers binaires compréhensibles par la machine (le processeur)
- **exécution** : tester le bon fonctionnement du programme
⇒ exploitation des capacités de l'appareil informatique et production de résultats

MNI 20 2011-2012

1 Introduction Langage C 1.2 Historique du C

1.2 Historique du C

- langage conçu dans les années 1970
- 1978 : parution de The C Programming Language de B. KERNIGHAN et D. RICHIE
- développement lié à la diffusion du système UNIX
- 1988–90 : normalisation **C89** ANSI-ISO (bibliothèque standard du C)
Deuxième édition du KERNIGHAN et RICHIE norme ANSI
- 1999 : norme **C99** en cours d'implémentation :
<http://gcc.gnu.org/c99status.html>
Ajout de nouveaux types (booléen, complexe, entiers de diverses tailles (prise en compte des processeurs 64 bits), caractères étendus (unicode), ...).
Introduction de la généricité dans les fonctions numériques,
déclarations tardives des variables, tableaux automatiques de taille variable...
⇒ se conformer à une **norme** pour la portabilité
- base d'autres langages dont **C++** (premier standard en 1998), **PHP**, **Java**, etc.

MNI 22 2011-2012

- L'ordinateur est muni d'un **système d'exploitation** (*exemple* : linux).
 - Le code source est un fichier texte écrit au moyen d'un **éditeur de texte**.
Exemples : vi, emacs, kate, kwrite, ... sous linux.
Un fichier code source C doit avoir une **extension .c**
 - Le code machine (fichier objet ou exécutable) est généré par un **compilateur**^a : programme qui analyse le code source, signale les erreurs de syntaxe, produit des avertissements sur les constructions suspectes, convertit un code source en code machine, optimise le code machine...
Exemples :
gcc (*GNU Compiler Collection* - compilateur C standard sous UNIX et linux),
icc (compilateur C d'Intel).
 - Les instructions du programme sont exécutées par un **processeur** caractérisé par son architecture, la taille de ses registres (nombre de bits traités ensemble : 32, 64 bits), sa vitesse d'horloge (mega ou giga Hertz), son jeu d'instructions...
Exemples : famille x86 d'Intel (32 bits), Pentium (32/64 bits) ou x64 (64 bits)
- a. Il existe des langages interprétés qui exécutent les instructions sans compilation (shell scripts).

MNI 21 2011-2012

1 Introduction Langage C 1.3 Intérêts du C

1.3 Intérêts du C

Langage C
langage de haut niveau (structures de contrôle, structures de données, fonctions, compilation séparée, ...)
mais aussi ... langage de bas niveau (manipulation de bits, d'adresses, ...)
applications scientifiques et de gestion
langage portable grâce à la norme et à des bibliothèques
langage puissant, efficace, mais aussi ... permissif !
écriture de systèmes d'exploitation

MNI 23 2011-2012

1.4 Généralités

	langage C
format	<<libre>> ⇒ mettre en évidence les structures par la mise en page (indentation)
ligne	pas une entité particulière (sauf préprocesseur) (la fin de ligne est un séparateur comme l'espace, la tabulation, ...)
fin d'instructions	instructions simples terminées par ;
commentaire	entre /* et */ (pas d'imbrication selon la norme) en C99 et C++ : introduit par // et terminé en fin de ligne
directive préprocesseur	Introduite par # en première colonne
Identificateur variable	commence par une lettre ou _ 31 caractères alphanumériques plus _ sont distingués
maj/minuscule	distinction

1.5 Exemple de programme C avec une seule fonction

```

/* programme elem0.c */
#include <stdio.h>           /* instructions préprocesseur */
#include <stdlib.h>         /* = directives
*/

int main(void)              /* fonction principale */
{                            /* <=< debut du bloc principal */
    int i ;                 /* déclaration */
    int s=0 ;               /* déclaration + initialisation*/

    for (i = 1 ; i <= 5 ; i++) /* structure de boucle */
    {                        /* <=< début de sous bloc */
        s += i ;
    }                        /* <=< fin de sous bloc */
    printf("somme des entiers de 1 à 5\n") ;
    printf("somme = %d\n", s) ;
    exit(0) ;               /* renvoie à unix le status 0 (OK) */
}                            /* <=< fin du bloc principal */

```

1.6 Structure générale d'un programme C

Structure générale d'un programme C élémentaire :

```

/* programme elem1.c */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    Déclarations des variables;

    Instructions exécutables;
}

```

Conseils pour la mise en page :

- une instruction par ligne
- indenter à l'ouverture d'un bloc

- Une **instruction simple** doit se terminer par ;
- Une **instruction composée** est constituée d'un **bloc** d'instructions (imbrication des blocs possible). Elle est délimitée par des accolades { et }.
- Un programme C = une (ou plusieurs) **fonction(s)** dont au moins la fonction **main** : le programme principal.
N.-B : à l'extérieur de ces fonctions, il peut comporter des instructions, des déclarations de variables, des déclarations de fonctions spécifiant leur prototype, et des directives pour le préprocesseur introduites par #.
- La définition d'une **fonction** se compose d'un **entête** et d'un **corps** (entre { et }) qui est en fait une instruction composée.
- L'**entête** d'une fonction spécifie le **type** de la valeur de retour, le nom de la fonction et ses paramètres ou arguments :
type nom_fonction (type1 arg1, type2 arg2, ...)
où chaque argument est déclaré par son type, suivi de son identificateur

1.7 Exemple de programme C avec deux fonctions

```

/* programme elem.c */
/* debut des instructions préprocesseur */
#include <stdio.h>           /* pour les entrées/sorties */
#include <stdlib.h>          /* par exemple pour exit */
/* fin des instructions préprocesseur */

int somme(const int p) ;      /* déclaration de la fonction somme */
int main(void)               /* fonction principale (sans param.)*/
{                             /* <=< début de bloc */
    int s ;                  /* déclaration de l'entier s */
    printf("somme des entiers de 1 à 5\n"); /* avec retour ligne => "\n" */
    s = somme(5) ;           /* appel de la fonction somme */
    printf("somme = %d\n", s) ; /* impression du résultat */
    exit(0) ;               /* renvoie à unix un status 0 (OK) */
}                             /* <=< fin de bloc */
/*                               */

```

1.8 Compilation

- Fichier ou code **source** (texte) de suffixe **.c** en C
- Fichier **objet** (binaire) de suffixe **.o**
- Fichier **exécutable** (binaire) **a.out** par défaut

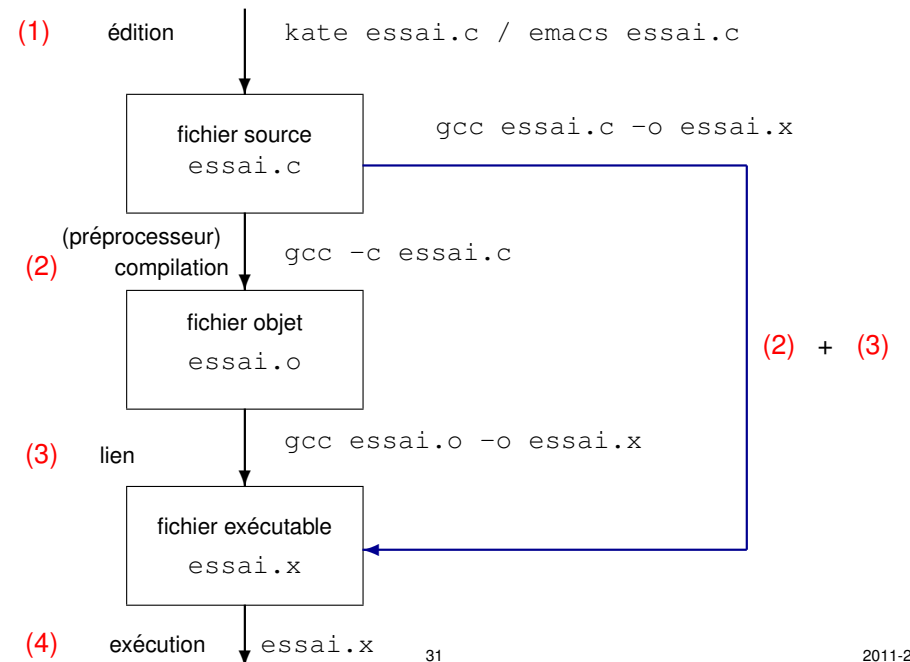
La commande de compilation **gcc toto.c** lance par défaut trois actions :

1. traitement par le **préprocesseur** (`cpp`) des lignes commençant par `#` (transformation textuelle) → fichier **texte** modifié
2. **compilation** à proprement parler → fichier **objet** `.o`
3. **édition de lien** (`gcc` lance `ld`) → fichier **exécutable** `a.out`

```

int somme(const int p)        /* définition de la fonction somme */
/*
 * calcul de la somme des p premiers entiers
 */
{                             /* <=< début de bloc */
    int i , sum ;            /* déclaration des var. locales */
    for (i = 0, sum = 0 ; i <= p ; i++) /* structure de boucle */
    {                         /* <=< début de bloc */
        sum += i ;           /* sum = sum + i */
        printf(" i = %d, somme partielle = %d\n", i, sum) ;
    }                         /* <=< fin de bloc */
    return sum ;             /* valeur rendue par la fonction */
}                             /* <=< fin de bloc */

```



Options de compilation permettant de choisir les étapes et les fichiers :

→ `gcc -E toto.c` : préprocesseur seulement

→ `-c` : préprocesseur et compilation seulement

→ `-o toto.x` : permet de spécifier le nom du fichier exécutable

→ `-ltruc` donne à `ld` l'accès à la **bibliothèque** `libtruc.a`

(ex. : `-lm` pour `libm.a`, bibliothèque mathématique indispensable en C)

Options de compilation utiles à la **mise au point** :

→ vérification des standards du langage (errors)

→ avertissements (warnings) sur les instructions suspectes (variables non utilisées, instructions apparemment inutiles, changement de type, ...)

→ vérification des passages de paramètres

(nécessite un contrôle interprocédural, donc les prototypes)

⇒ faire du compilateur un **assistant efficace** pour **anticiper les problèmes** avant l'édition de lien ou, pire, l'exécution.

1.9 Compilateur

langage C
gcc (dans le shell)
avec options sévères
C89 (ANSI) → alias gcc-mni-c89
C99 → alias gcc-mni-c99
doc : http://gcc.gnu.org

2 Types des variables

Le C est un langage typé : il faut déclarer chaque variable utilisée dans le code ⇒ indiquer quel est le type de la variable utilisée.

2.1 Types de base

Type	Dénomination
vide	<code>void</code>
booléen	(C99) bool <code>#include <stdbool.h></code>
Entier	
caractère	<code>char</code>
caractère large	(C99) <code>wchar_t</code>
court	<code>short (int)</code>
courant	<code>int</code>
long	<code>long (int)</code>
plus long	(C99) long long
Réel	
simple précision	<code>float</code>
courant	<code>double</code>
quadruple précision	<code>long double</code>
(C99) Complexe	
simple	<code>float complex</code>
courant	<code>(double) complex</code>
long	<code>long double complex</code>
	<code>#include <tgmath.h></code>

2.2 Déclaration et affectation des variables

Déclarer une variable = réserver une zone en mémoire pour la stocker

Chaque variable déclarée est stockée dans une zone mémoire qui lui est propre (mémoire vive ou **RAM**) sous un certain codage :

- emplacement de cette zone : **adresse unique** propre à chaque variable.
- taille de cette zone : dépend du **type** de la variable (et du processeur 32/64 bits).

C89 : déclarer en tête des fonctions (**syntaxe conseillée pour les débutants**)

C99 : n'importe où (**mais en tête de bloc pour la lisibilité du code source**)

Affecter une variable = stocker une valeur dans la zone mémoire réservée

Initialiser une variable = affecter une valeur à une variable au moment de la réservation de la mémoire

Remarque importante : avant initialisation (ou première affectation) la valeur d'une variable est indéterminée.

2.2.1 Syntaxe et exemples

```
type identifiant1, identifiant2=valeur ... ;
```

Exemples avec des entiers :

- Déclaration de 3 entiers : `int j, j2, k_max ;`
- Déclaration avec initialisation : `int a = 2, b = 3 ;`
- Affectation (après déclaration) : `k_max=4 ;`

Exemples avec des floats :

- Déclaration de 3 floats : `float x, y, z ;`
- Déclaration avec initialisation : `float v = 1.5f, w = 3.f ;`
- Affectation (après déclaration) : `z=4.5f ;`

Exemples avec des doubles :

- Déclaration de 3 doubles : `double x, y, z ;`
- Déclaration avec initialisation : `double v = 1.5, w = 3. ;`
- Affectation (après déclaration) : `z=4.5 ;`

2.2.2 Valeurs

Type	langage C
bool	(C99) true false
char	'a'
Chaînes	"chaîne" "\n" "s'il"
short	17
int (décimal)	17
int (octal)	021 (attention)
int (hexadécimal)	0x11
long	17 l
long long	17 ll
float	-47.1 f -6.2e-2 f
double	-47.1 -6.2e-2
long double	-47.1 l -6.2e-2 l
complex	(C99) 3.5+I*2.115

2.3 Domaine et représentation machine des entiers

2.3.1 Domaine des entiers non-signés et signés

Attributs **unsigned** et **signed** des types entiers ou caractères pour indiquer si le bit de poids fort est un bit de signe (cas par défaut pour les entiers).

taille et **domaine** des entiers (dépend de la machine) en base 2

type	taille	unsigned	signed
char	1 octet	0 → 255 = 2 ⁸ - 1	-128 → +127
short	2 octets	0 → 2 ¹⁶ - 1	-2 ¹⁵ → 2 ¹⁵ - 1
int	4 octets	0 → 2 ³² - 1	-2 ³¹ → 2 ³¹ - 1
long	(4 ^a ou 8 ^b octets)	0 → 2 ⁶⁴ - 1	-2 ⁶³ → 2 ⁶³ - 1

a. Machine 32 bits

b. Machine 64 bits

Pour les entiers signés : (en base 2 et base 10)

Rappel : $\log_{10} 2 \approx 0,30 \Rightarrow 2^{10} = 1024 = 10^{10 \log_{10}(2)} \approx 10^3$

	C	
sur 32 bits = 4 octets	INT_MAX	2 ³¹ ≈ 2 × 10 ⁹
sur 64 bits = 8 octets	LONG_MAX	2 ⁶³ ≈ 8 × 10 ¹⁸

⇒ C99 : types entiers étendus à nb d'octets imposé ou à minimum imposé
par exemple : `int32_t` ou `int_least64_t`

Dépassement de capacité en entier ⇒ passage en négatif

2.3.2 Représentation machine des entiers

Stockage en mémoire ⇒ **représentation binaire**

(représentation exacte pour les entiers mais pas pour les flottants, en général)

Représentation binaire dans le cas des entiers :

- **entier non signé** : décomposition de l'entier en base 2
- **entier signé** :
 - bit de poids fort : 0 (entier positif) ou 1 (entier négatif),
 - bits restants : décomposition de l'entier en base 2 (complémentaire bit à bit + 1 pour les entiers négatifs)

Exemples :

Entier non-signé $i=10$ codé sur 4 octets : 00000000000000000000000000001010.

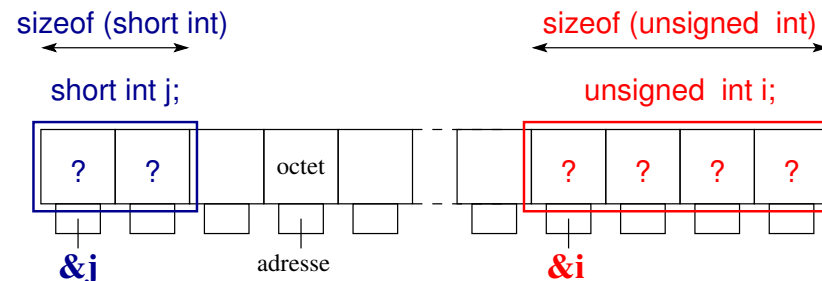
Entier signé court $j=-1$ codé sur 2 octets : 1111111111111111.

2.3.3 Structure simplifiée de la RAM : déclaration de deux entiers

Déclaration de deux variables :

```
short int j; unsigned int i;
```

Structure simplifiée de la RAM **au cours de l'exécution de ces instructions** :

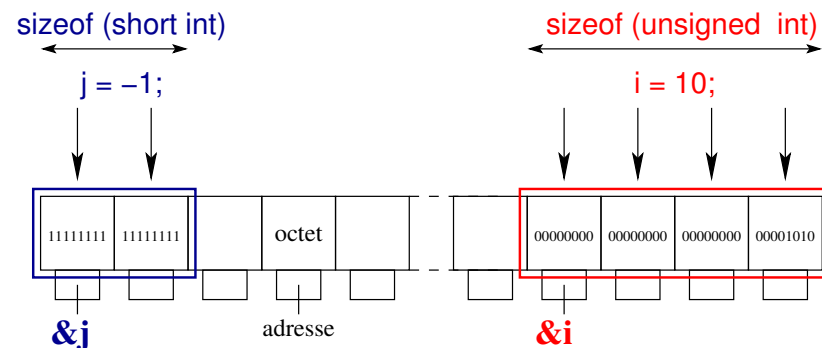


2.3.4 Structure simplifiée de la RAM : affectation de deux entiers

Affectation des deux variables (après déclaration) :

```
j=-1; i=10;
```

Structure simplifiée de la RAM **au cours de l'exécution de ces instructions** :



De manière générale :

- **la mémoire est segmentée** (segment élémentaire : 1 octet)
- **chaque segment est associé à une adresse** (nombre entier long non-signé)

En particulier, lors de la compilation de l'instruction de déclaration :

- réservation d'un nombre d'octets qui dépend du type de la variable (`sizeof (type)` donne la **taille** en octets du type `type`)
- attribution d'une **adresse** : celle associée au premier segment occupé (`&i` et `&j` sont les adresses de `i` et `j` où `&` est l'opérateur adresse)
- si la variable n'a pas été initialisée sa valeur est indéterminée (?)

2.4 Valeurs maximales des entiers en C

```

/* programme limites-int-machine.c */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h> /* valeurs limites definies ici */
int main(void)
{
    /* impression des valeurs limites des entiers sur la machine */
    /* non-signés puis signés en décimal, hexadécimal et octal */

    /* entier long */
    printf("%-18s %20lu %16lx %22lo\n",
           "Unsigned-Long-max", ULONG_MAX, ULONG_MAX, ULONG_MAX);
    printf("%-18s %20ld %16lx % 22lo\n",
           "Long-max", LONG_MAX, LONG_MAX, LONG_MAX);

```

```

/* entier */
printf("%-18s %20u %16x % 22o\n",
       "Unsigned-Int-max", UINT_MAX, UINT_MAX, UINT_MAX);
printf("%-18s %20d %16x % 22o\n",
       "Int-max", INT_MAX, INT_MAX, INT_MAX);

/* entier court */
printf("%-18s %20hu %16hx % 22ho\n",
       "Unsigned-Short-max", USHRT_MAX, USHRT_MAX, USHRT_MAX);
printf("%-18s %20hd %16hx % 22ho\n",
       "Short-max", SHRT_MAX, SHRT_MAX, SHRT_MAX);
exit(0);
}

```

MNI

44

2011-2012

MNI

45

2011-2012

2 Types des variables

Langage C

2.4 Valeurs maximales des entiers en C

2.4.1 Valeurs maximales des entiers en C : machine 32 bits

```

Unsig-Lng-Lng-max 18446744073709551615 ffffffffffffffff 17777777777777777777
Long-Long-max    9223372036854775807 7fffffffffffffff 77777777777777777777

```

Unsigned-Long-max	4294967295	fffffff	3777777777
Long-max	2147483647	7fffffff	1777777777
Unsigned-Int-max	4294967295	fffffff	3777777777
Int-max	2147483647	7fffffff	1777777777
Unsigned-Short-max	65535	ffff	177777
Short-max	32767	7fff	77777

2.4.2 Valeurs maximales des entiers en C : machine 64 bits

```

Unsig-Lng-Lng-max 18446744073709551615 ffffffffffffffff 17777777777777777777
Long-Long-max    9223372036854775807 7fffffffffffffff 77777777777777777777

```

Unsigned-Long-max	18446744073709551615	fffffffffffffff	17777777777777777777
Long-max	9223372036854775807	7fffffffffffffff	77777777777777777777
Unsigned-Int-max	4294967295	fffffff	3777777777
Int-max	2147483647	7fffffff	1777777777
Unsigned-Short-max	65535	ffff	177777
Short-max	32767	7fff	77777

MNI

46

2011-2012

MNI

2.5 Domaine et précision des flottants

2.5.1 Codage des flottants

Un flottant est un nombre réel qui est décrit par (en décimal ici, en machine le codage est en binaire) :

- **un signe**
- **une mantisse** dont le nombre maximal de digits indique la précision
- **un exposant** dont le nombre maximal de digits indique le domaine.

Lorsque le nombre maximal de digits est fixé et que l'exposant peut varier on parle de représentation en **virgule flottante**.

Exemples en base 10 : (la mantisse m est telle que : $1 \leq m < 10$)

- 100,5 correspond à $1,005 \times 10^2$
⇒ exposant de 2 (1 digit) et mantisse $m = 1,005$ (3 digits)
- 1000000020 correspond à $1,000000020 \times 10^9$
⇒ exposant de 9 (1 digit) et mantisse $m = 1,000000020$ (9 digits)

En machine, un flottant codé sur 4 octets est décomposé en :

1 bit de signe, 23 bits pour la mantisse et 8 bits pour l'exposant.

2011-2012

2.5.2 Domaine et précision des flottants

- **Domaine** fini comme pour les entiers : valeur finie de l'exposant
- **Précision** limitée contrairement aux entiers : nombre fini de digits de la mantisse (ε défini comme la plus grande valeur telle que $1 + \epsilon = 1$)

⇒ les flottants ne peuvent être représentés exactement, en général.

simple préc. = 4 octets	FLT_MAX	$3,4 \times 10^{38}$
	FLT_MIN	$1,18 \times 10^{-38}$
	FLT_EPSILON	$1,2 \times 10^{-7}$
double préc. = 8 octets	DBL_MAX	$1,8 \times 10^{308}$
	DBL_MIN	$2,2 \times 10^{-308}$
	DBL_EPSILON	$2,2 \times 10^{-16}$

Application :

Pas de différence entre 1000000020 et 1000000000 pour des variables du type float puisque la mantisse est de 9 digits alors que le type float autorise une précision de 7 digits au maximum.

2.6.2 Exemples d'attribut const en C

```
/* programme const.c */
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    const int i = 2 ; /* non modifiable */

    i++ ; /* => erreur à la compilation */
    printf("i vaut %d\n", i);

    exit(0);
}
```

Avec gcc par exemple

const.c:9: **error:** increment of read-only variable 'i'

2.6 Constantes

2.6.1 Syntaxe

Attribut **const** devant le type

```
const int j = -1;
```

mais la non-modification des constantes (via une fonction notamment) n'est pas toujours respectée ⇒ on pourra également utiliser la directive **#define** du préprocesseur

```
#define J -1 /* attention: pas de ; */
```

Le préprocesseur substituera J par -1 partout dans le programme... mais la constante n'est plus typée

2.6.3 Exemples d'utilisation de #define

```
/* programme const2.c */
#include <stdio.h>
#include <stdlib.h>
#define J 2 /* préproc. remplace J par 2 */

int main(void) {
    J=1 ; /* => erreur à la compilation */

    exit(EXIT_SUCCESS);
}
```

Avec gcc par exemple

const2.c:5: invalid lvalue in assignment

3 Opérateurs

Opérateurs unaires : agissent sur un seul argument.

Opérateurs binaires : agissent sur deux arguments.

Opérateurs ternaires : agissent sur trois arguments.

3.1 Opérateur d'affectation

Opérateur binaire évalué de droite à gauche :

lvalue = expression ⇒ conversions implicites éventuelles

(le terme de droite est converti dans le type du terme de gauche)

En cas de conversion il est fortement conseillé de l'expliquer :

lvalue = (type) expression ⇒ conversion explicite (opérateur **cast**)

Attention aux problèmes d'**étendue/précision** lors de l'affectation.

3.2 Opérateurs algébriques

	langage C
addition	+
soustraction	-
multiplication	*
division	/
reste modulo	%

Remarques :

- opérateurs +, -, *, / et % : opérateurs binaires agissant de gauche à droite.
- attention à la différence entre les opérateurs algébriques binaires – et + d'une part, et les opérateurs unaires de signe : – (opposé) et + d'autre part.
- élévation à la puissance au moyen de la fonction **pow(x, y)**
(inclure le fichier **tgmath.h** dans le code source et ajouter l'option **-lm** à la compilation).

```

/* programme precision.c */
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int a=123456789;
    float b=0.123456789f;
    float c;
    printf("float: %d\nint: %d\n", sizeof(float), sizeof(int));
    c=(float)a;
    printf("%d %.10g %.10g\n", a, c, b); /* passage en double
                                         pour plus de précision - attention au f*/
    exit(EXIT_SUCCESS);
}

```

Résultat à l'exécution :

```

float : 4
int   : 4
123456789 123456792 0.123456791

```

3.3 Opérateurs de comparaison

3.4 Opérateurs logiques

résultat	entier
inférieur à	<
inférieur ou égal à	<=
égal à	==
supérieur ou égal à	>=
supérieur à	>
différent de	!=

ET	&&
OU	
NON	!

Attention : ne pas confondre l'opérateur test d'égalité == avec l'opérateur d'affectation =.

3.12 Priorités des opérateurs en C

- opérateurs sur les tableaux, fonctions, structures : `[]`, `()`, `->`, `.`
 - opérateurs unaires `+`, `-`, `++`, `--`, `!`, `~`, `*`, `&`, `sizeof`, (cast)
 - opérateurs algébriques `*`, `/`, `%`
 - opérateurs algébriques `+`, `-`
 - opérateurs de décalage `<<`, `>>`
 - opérateurs relationnels `<`, `<=`, `>`, `>=`
 - opérateurs relationnels `==`, `!=`
 - opérateurs sur les bits `&`, puis `^`, puis `|`
 - opérateurs logiques `&&`, puis `||`
 - opérateur conditionnel `?` :
 - opérateurs d'affectation `=` et les affectations composées
 - opérateur séquentiel `,`
- ⇒ indiquer les priorités avec des parenthèses !

```

/* programme printf_scanf.c */
#include <stdio.h> /* contient printf et scanf */
#include <stdlib.h>

int main(void) {
    int i;
    float x;
    double y;
    printf("Entrer un entier\n");
    /* Ne pas oublier l'opérateur adresse dans scanf ! */
    scanf("%d", &i);
    printf("La valeur de i est %d\n", i);

    printf("Entrer deux réels: float, double\n");
    scanf("%g %lg", &x, &y);
    /* Différence de format scanf/printf pour les flottants ! */
    printf("Les valeurs de x et y sont %g et %g\n", x, y);

    exit(EXIT_SUCCESS);
}

```

4 Entrées et sorties standard élémentaires

4.1 Généralités

Ecriture sur `stdout` = écran :

```
printf("format", liste d'expressions)
```

⇒ afficher à l'écran (`stdout`) des messages et les valeurs des variables

Lecture depuis `stdin` = clavier :

```
scanf("format", liste d'adresses)
```

⇒ lire du clavier (`stdin`) les valeurs des variables

⇒ stocker ces valeurs aux **adresses** spécifiées par les arguments

Attention : opérateur adresse `&` dans `scanf` (en général).

Format : spécifier le type par `%` de chaque variable (gabarit optionnel)

Spécifier `\n` en sortie pour changer de ligne

4.2 Formats d'affichage en C

Attention : quelques différences entre `scanf` (type exact) et `printf` (conversion de type possible)

En sortie `printf`

Type	Format
char	<code>%c</code>
chaîne	<code>%s</code>
int/short	<code>%d</code>
unsigned int/unsigned short	<code>%u</code> (<code>%o</code> , <code>%x</code>)
long	<code>%ld</code>
unsigned long	<code>%lu</code> (<code>%lo</code> , <code>%lx</code>)
long long	<code>%lld</code>
unsigned long long	<code>%llu</code>
float/double	(<code>%e</code> , <code>%f</code>) <code>%g</code>
long double	(<code>%le</code> , <code>%lf</code>) <code>%lg</code>

En entrée scanf

Type	Format
char	%c
short	%hd
unsigned short	%hu (%ho, %hx)
int	%d
unsigned int	%u (%o, %x)
long	%ld
signed long	%lu (%lo, %lx)
long long	%lld
unsigned long long	%llu (%llo, %llx)
float	(%e, %f) %g
double	(%le, %lf) %lg
long double	(%Le, %Lf) %Lg

5 Structures de contrôle

Par défaut : exécution des instructions une fois, dans l'ordre dans lequel elles apparaissent dans le code source

⇒ trop restrictif.

Structures de contrôle (flow control en anglais) : permettent de modifier le cheminement lors de l'exécution des instructions.

Différents types de structures de contrôle :

- exécution conditionnelle (**if**) ou aiguillage (**switch**) dans les instructions,
- itération de certains blocs (**while**, **for**)
- branchements (**break**, **continue**)

Elles peuvent être **combinées** au sein d'un même programme.

4.3 Gabarits d'affichage en C

4.3.1 Cas des entiers

Structure générale : **%wd** où **w** est le gabarit d'affichage.

Le gabarit est un nombre qui indique la largeur minimale du champ d'affichage.

Exemple : **%5d** → au moins 5 caractères sont réservés à l'affichage de l'entier.

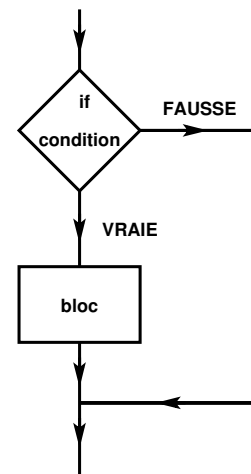
4.3.2 Cas des flottants

Structure générale : **%w.pf** où **w** indique la largeur minimale du champ d'affichage (incluant le point décimal) dont **p** caractères sont réservés à la partie décimale (précision).

Exemple : **%5.3f** → 5 caractères sont réservés à l'affichage du flottant dont 3 pour la partie décimale.

5.1 Structure if

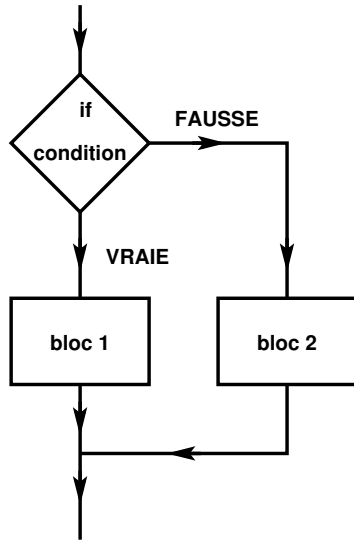
Permet de **choisir** quelles instructions vont être exécutées :



```

if (expression) {
    bloc d'instructions;
    /* si l'expression est vraie */
}
  
```

(expression :
 - vraie si elle est !=0
 - fausse si elle est ==0)

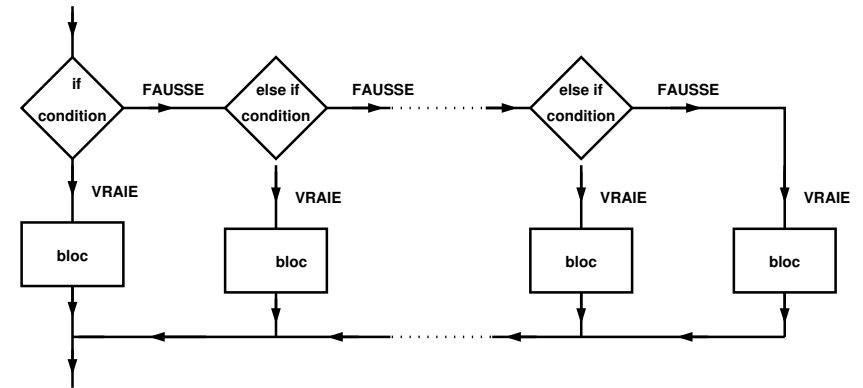


```

if(expression) {
    bloc d'instructions 1;
    /* si l'expression est vraie */
}
else {
    bloc d'instructions 2;
    /* si l'expression est fausse */
}

```

Des **if** {...} **else** {...} peuvent être imbriqués :



```

if (expression_1) {
    /* si expression_1 est vraie */
}
else {
    if (expression_2) {
        /* si expression_1 est fausse et expression_2 est vraie */
    }
    else {
        /* si expression_1 et expression_2 sont fausses */
    }
}

```

Attention dans ce cas à bien **respecter l'indentation** pour montrer la structuration du programme.

5.1.1 Exemple de if

```

/* programme if.c */
#include <stdio.h>
#include <stdlib.h>
/* structure if ... else
   affichage du max de deux nombres */
int main(void) {
    int i, j, max;
    printf("entrer i et j (entiers)\n");
    scanf("%d %d", &i, &j);

    if (i >= j) { /* bloc d'instructions */
        printf(" i >= j \n");
        max = i;
    }
}

```

```

else {      /* bloc d'instructions */
    max = j;
}
printf(" i= %d, j= %d, max = %d\n", i, j, max);

exit(0);
}

```

MNI

72

2011-2012

5.2 Structure switch (pas avec des flottants)

Aiguillages multiples :

```

switch (expression_entière) {
    case sélecteur1 :
        instructions; /* si expression == selecteur1 */
        break;        /* optionnel */
    case sélecteur2 :
        instructions; /* si expression == selecteur2 */
        break;        /* optionnel */
    ...
    default :         /* optionnel */
        instructions; /* dans tous les autres cas */
}

```

sélecteur : une expression **constante** entière ou caractère (ex : 3 ou 'z')

Si on ne précise pas **break**, on passe par toutes les instructions **suivant** le cas sélectionné.

MNI

73

2011-2012

5.2.1 Exemples de switch-case

```

/* programme case.c */
#include <stdio.h>
#include <stdlib.h>

/* structure switch case */
int main(void)
{
    int i ;
    printf(" entrer un entier : ");
    scanf("%d", &i);
    printf("\n i = %d \n", i);
    switch (i) { /* début de bloc */
        case 0 :
            printf(" i vaut 0 \n");

```

```

        break; /* nécessaire ici ! */
        case 1 :
            printf(" i vaut 1 \n");
            break; /* nécessaire ici ! */
        default :
            printf(" i différent de 0 et de 1 \n");
    } /* fin de bloc */
    exit(0);
}

```

MNI

74

2011-2012

MNI

75

2011-2012

```

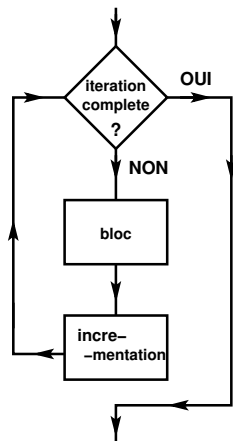
/* programme casel.c */
#include <stdio.h>
#include <stdlib.h>
/* exemple d'utilisation de la structure case sans break
 * pour "factoriser des cas" et les traiter en commun
 */
int main(void)
{
    char c ;
    printf("entrer un caractère: est-ce une ponctuation double ?");
    scanf("%c", &c);
    printf("\n caractère = %c \n", c);
    switch (c) {
        case '?' :
        case '!' :
        case ';' :

```

5.3 Structures itératives ou boucles

Elles permettent de répéter plusieurs fois un bloc d'instructions.

5.3.1 Boucle définie (for)



Quand le nombre de répétitions est connu, on utilise **for** :

```

for (expr-1; expr-2; expr-3) {
    instructions ;
}

```

```

        case ':' :
            printf("ponctuation double \n");
            break ;
        default :
            printf("autre caractère \n");
    }
    exit(0) ;
}

```

boucle	for (expr-1; expr-2; expr-3) { instructions ; }
--------	--------------------------------------------------------------

- expr-1 est effectué **une fois** avant l'entrée dans la boucle (généralement initialisation d'un compteur de tours)
- expr-2 est une condition "tant que", évaluée à chaque début de répétition (généralement test sur le compteur de tour)
- expr-3 est effectué à la fin de chaque répétition (généralement incrémentation du compteur de tours)

5.3.2 Exemple de boucle for

```

/* programme for.c */
#include <stdio.h>
#include <stdlib.h>
/* affichage des entiers impairs inférieurs à un entier donné
   mise en oeuvre de la structure "for" */
int main(void) {
    int i, m = 11;
    printf("affichage entiers impairs <= %d \n", m);

    for (i = 1; i <= m; i = i + 2) { /*bloc d'instructions repetees*/
        printf("%d \n", i);
    }
    exit(0);
}

```

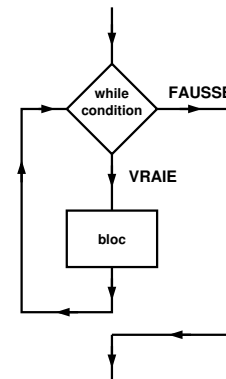
tant que	while (expr) { instructions ; }
faire ...	do { instructions ;
tant que	} while (expr) ;

- Dans le cas de la boucle **while** :
les instructions sont répétées "tant que" expr est vraie.
- Dans le cas de la boucle **do...while** :
les instructions sont exécutées **au moins** une fois,
puis répétées "tant que" expr est vraie.

Attention aux boucles infinies si expr est toujours vraie.

5.3.3 Boucle indéfinie (while et do...while)

Quand le nombre de répétitions (itérations) est a priori inconnu, on utilise **while** (ou **do...while**) :



```

while (expr) {
    instructions ;
}

```

5.3.4 Exemple de boucle while

```

/* programme while.c */
#include <stdio.h>
#include <stdlib.h>
/* mise en oeuvre de la structure "while" */
int main(void) {
    int i, m = 11 ;
    printf("affichage entiers impairs <= %d \n", m);
    i = 1 ;
    while ( i <= m ) { /* bloc d'instructions repetees*/
        printf(" %d \n", i);
        i += 2;
    }
    exit(0) ;
}

```

5.4 Branchements

Les branchements permettent de modifier le comportement des boucles.

bouclage anticipé	<code>continue;</code>
sortie anticipée	<code>break;</code>
branchement	<code>goto étiquette;</code>

L'étiquette est un identificateur suivi de `:` en tête d'instruction.

5.4.1 Exemple de continue

```
/* programme recycle.c */
#include <stdio.h>
#include <stdlib.h>
/* recyclage anticipé via "continue" */
int main(void)
{
    int i = 0 , m = 11;
    while ( i < m ) {
        i++;
        if ((i % 2) == 0 ) continue ; /* rebouclage si i pair */
        printf("%d \n", i) ;
    }
    exit(0) ;
}
```

5.4.2 Exemple de break

```
/* programme break.c */
#include <stdio.h>
#include <stdlib.h>
int main(void){ // utilisation de break
    int i = 1, m = 11 ;
    while (1) { /* a priori boucle infinie */
        printf(" %d \n", i);
        i += 2;
        if (i > m) {
            break; // sortie de la boucle
        }
    }
    exit(0) ;
}
```

6 Introduction aux pointeurs

6.1 Intérêt des pointeurs

Motivation :

En C, les pointeurs sont **indispensables**, notamment pour leur utilisation en lien avec les fonctions (mais aussi tableaux, allocation dynamique, etc.)

Définition :

Les pointeurs sont des variables **contenant l'adresse** d'autres variables d'un type donné.

Finalité :

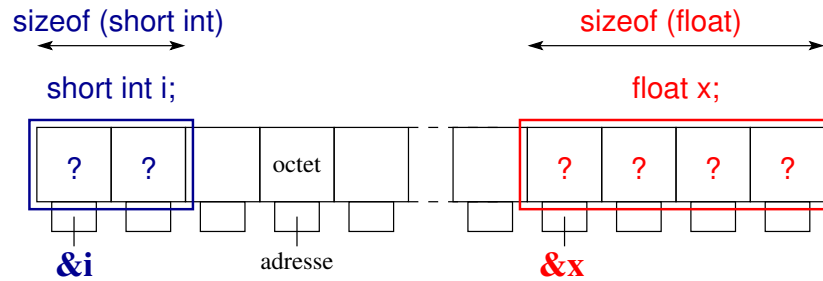
Un pointeur permet d'agir **indirectement** sur une variable : *via son adresse*, au lieu d'agir **directement** sur la variable : *via son identifiant*.

6.2 Déclaration et affectation

6.2.1 Déclaration d'une variable ordinaire (rappel)

`short int i; float x;` ⇒ réservation d'une **zone mémoire** :

- sa **taille** : dépend du type de la variable déclarée
(`sizeof(type)` : taille en octets du type `type`)
- son **emplacement** : adresse du premier octet sur lequel la variable est stockée
(`&var` : adresse de la variable `var`)



6.2.3 Déclaration d'un pointeur

`short int *pti; float *ptx;`

= réservation d'une zone mémoire pour stocker **l'adresse de la variable pointée**

Remarques :

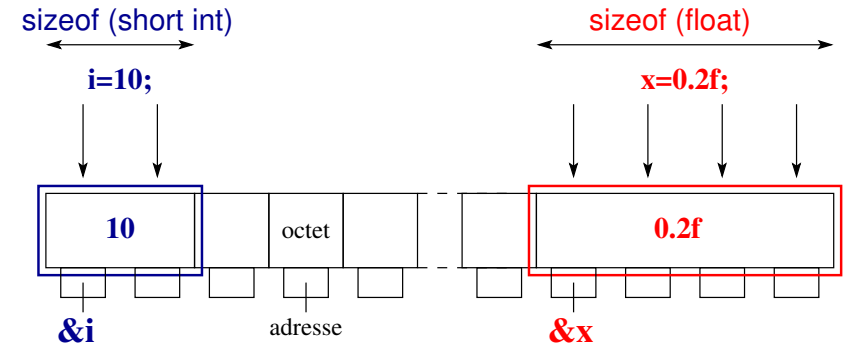
- comme toute autre variable un pointeur possède lui-même une adresse
- contrairement aux variables ordinaires la taille d'un pointeur est fixe
(elle ne dépend pas du type de la variable pointée)

6.2.2 Affectation d'une variable ordinaire (rappel)

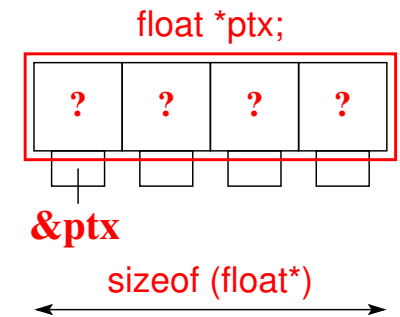
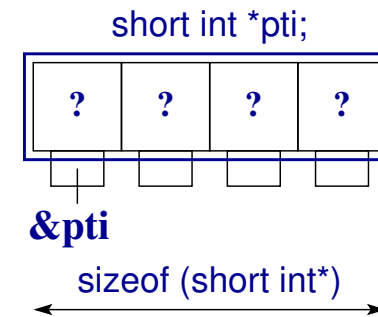
`i=10; x=0.2f;`

= stocker la valeur dans la zone mémoire réservée.

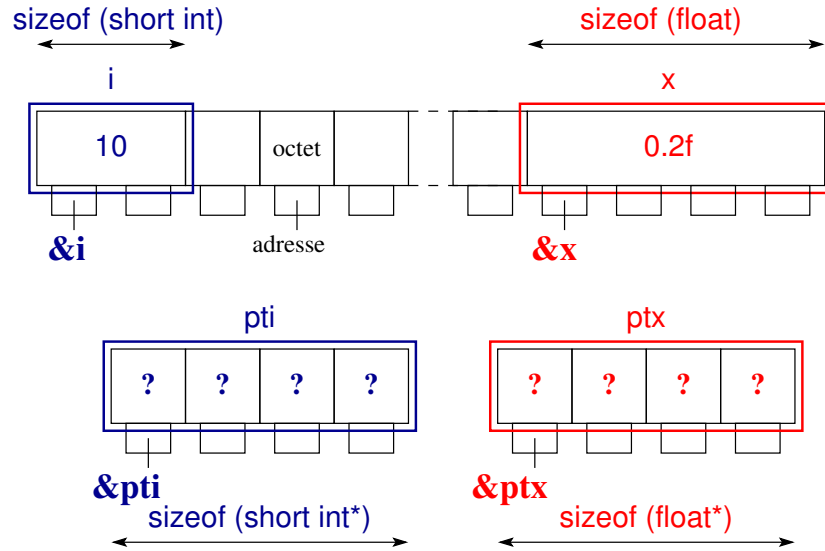
Attention : avant initialisation (= première affectation) la valeur d'une variable est indéterminée



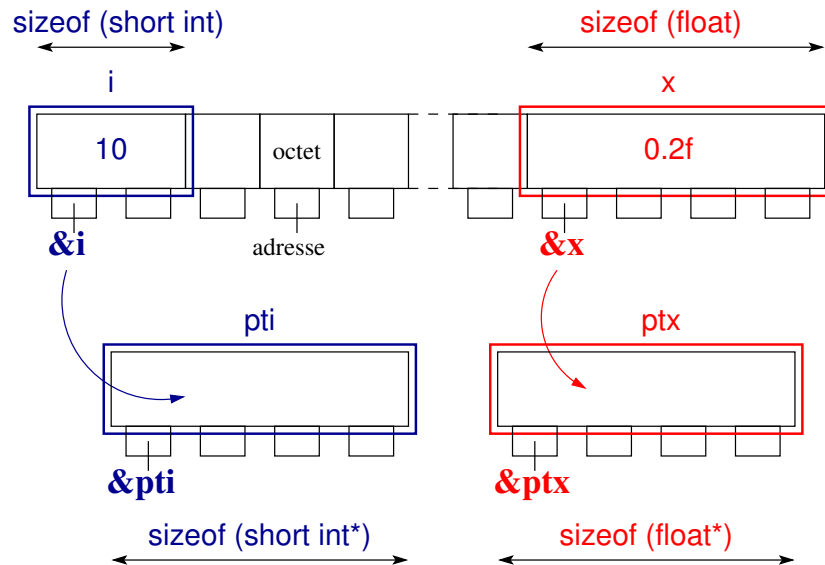
Pour simplifier : représentation binaire des variables stockées n'est pas montrée.



Visualisation des pointeurs et des variables ordinaires :



Visualisation de l'affectation des pointeurs : `pti=&i; ptx=&x;`



6.2.4 Affectation d'un pointeur

Affecter une adresse à un pointeur :

```
pti=&i; ptx=&x;
```

= stocker l'adresse mémoire d'une variable dans la zone mémoire réservée lors de la déclaration du pointeur.

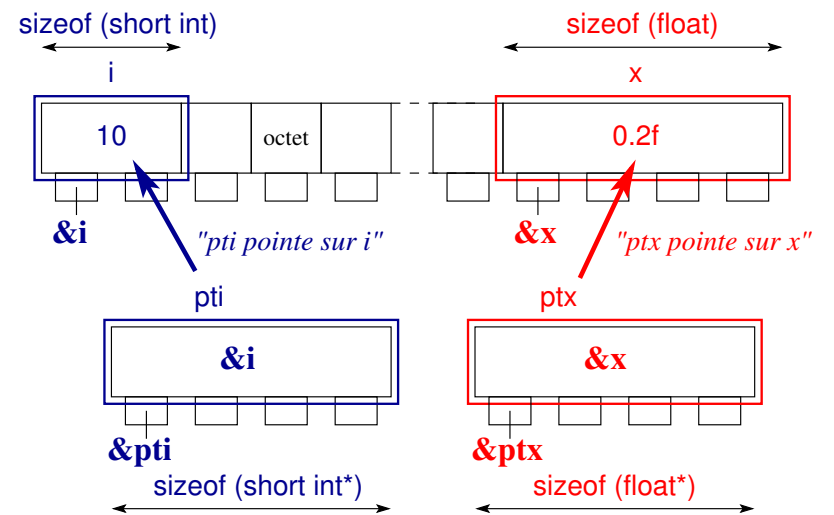
On dit que :

- (le pointeur) **pti pointe sur** (la variable) **i**,
- (le pointeur) **ptx pointe sur** (la variable) **x**.

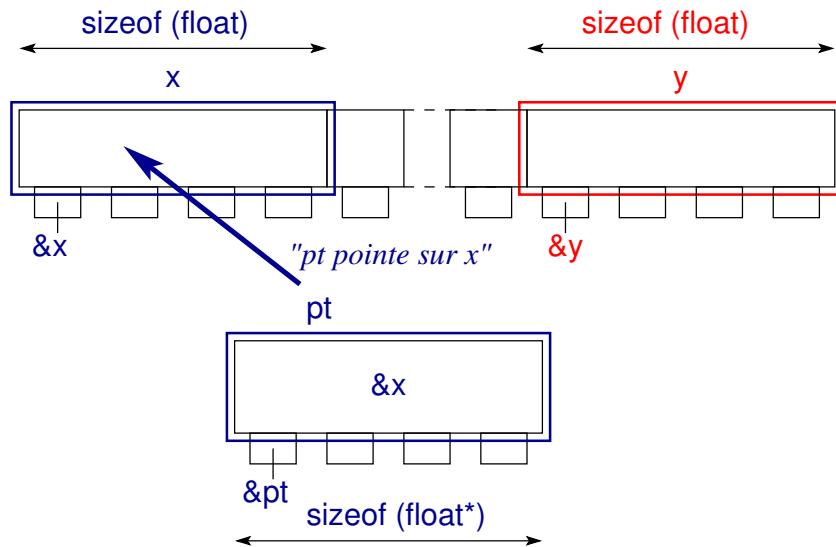
Attention :

- il faut que les variables `i` et `x` aient été déclarées au préalable.
- comme pour une variable ordinaire l'adresse contenue dans le pointeur est indéterminée avant l'initialisation du pointeur
⇒ **initialiser un pointeur avant de le manipuler** (cf. dissociation)

Visualisation après affectation (`pti` pointe sur `i` et `ptx` pointe sur `x`) :



Affectation d'un pointeur de float : `pt=&x;` (pt pointe sur x)



6.3 Indirection (opérateur *)

L'indirection est l'opération permettant d'**accéder au contenu de la variable pointée** via le pointeur (donc indirectement).

```
*pti=4;
```

La variable sur laquelle pointe `pti` vaut désormais 4.

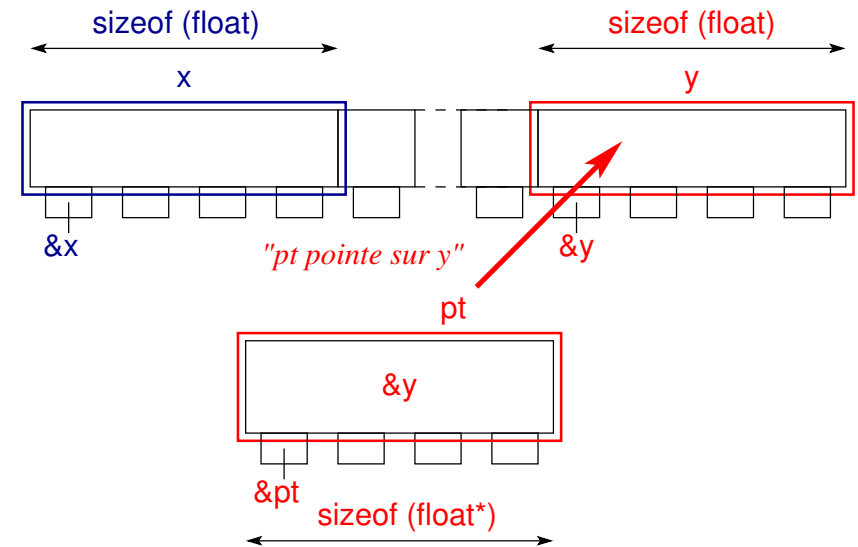
Remarque : cette opération est désastreuse si le pointeur n'a pas été initialisé
 ⇒ modification non-volontaire d'une variable, accès à une zone mémoire interdite (**segmentation fault**)

La déclaration d'un pointeur :

```
double *ptd;
```

peut se lire ***ptd** est un **double**

Re-affectation d'un pointeur de float : `pt=&y;` (pt pointe sur y)



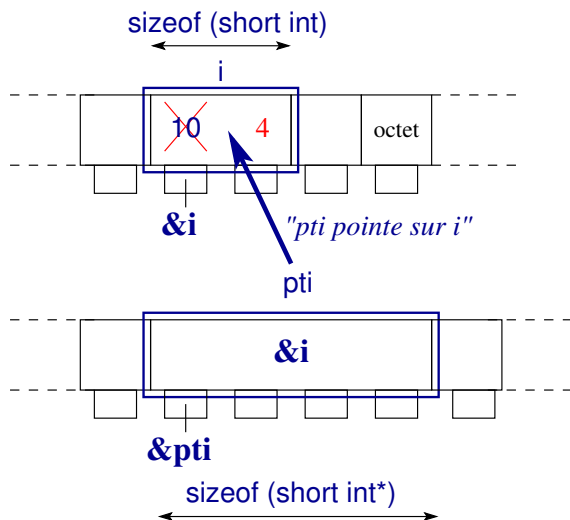
```
/* programme indirection.c */
#include <stdio.h>
#include <stdlib.h>
/* modification de la valeur d'une variable */
int main(void) {
    short int i;          /* declaration de i */
    short int *pti=&i;    /* declaration + affectation de pti */

    i=10; /* initialisation de i par acces direct */
    printf("La valeur initiale de i est %d\n", i);

    *pti=4; /* modification indirecte de la valeur de i */
    printf("La valeur finale de i est %d\n", i);

    exit(EXIT_SUCCESS);
}
```

Visualisation de la modification indirecte de la valeur d'une variable (`*pti=4;`):



6.4 Initialisation des pointeurs et dissociation

Déclarer un pointeur sans l'initialiser peut entraîner des erreurs.

La dissociation permet de dissocier un pointeur de la variable vers laquelle il pointe :

```
*pt=NULL;
```

⇒ à utiliser lors de l'initialisation par défaut des pointeurs.

```
/* programme erreur-pointeur.c */
/* exemple de pointeur non initialisé => erreur d'accès mémoire */
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int i;
    /* suivant l'ordre *pi, *pj => ? mais *pj, *pi => pj=0 erreur */
    int *pj, *pi;
    pi = &i; /* pi devient l'adresse de l'entier i */
    /* affichage des valeurs des pointeurs pi et pj */
    printf("pi=%lu, pj=%lu\n", (unsigned long int) pi,
           (unsigned long int) pj);
    /* accès à une zone interdite quand on affecte 2 à l'adresse pj */
    *pj = 2; /* segmentation fault */
    i = 1;
    printf("i= %d, *pi= %d , *pj= %d\n", i, *pi, *pj);
    exit(EXIT_SUCCESS);
}
```

6.5 Bilan concernant la syntaxe des pointeurs

<code>type *ptr;</code>	déclaration sans initialisation (déconseillé)
<code>type *ptr=NULL;</code>	déclaration avec initialisation (conseillé)
<code>type var;</code>	cible
<code>type *ptr=&var;</code>	déclaration avec initialisation (conseillé à condition que la cible ait déjà été déclarée)
<code>ptr = &var ;</code>	ptr pointe sur var
<code>*ptr</code>	valeur de la variable pointée par ptr
<code>ptr = NULL ;</code>	dissocier

6.6 Tailles des types de base et adresses en C

----- Processeur 32 bits AMD -----

Tailles en octets via sizeof :

```
char = 1 short int = 2 int = 4 long int = 4
```

Adresses converties en unsigned long int :

```
c =3219368569 s =3219368562 i =3219368548 l =3219368536
&c[0]=3219368569 &s[0]=3219368562 &i[0]=3219368548 &l[0]=3219368536
&c[1]=3219368570 &s[1]=3219368564 &i[1]=3219368552 &l[1]=3219368540
&c[2]=3219368571 &s[2]=3219368566 &i[2]=3219368556 &l[2]=3219368544
```

Tailles en octets via sizeof :

```
float = 4 double = 8 long double = 12 int * = 4
```

Adresses converties en unsigned long int :

```
f =3219368524 d =3219368496 L =3219368448 pi =3219368436
&f[0]=3219368524 &d[0]=3219368496 &L[0]=3219368448 &pi[0]=3219368436
&f[1]=3219368528 &d[1]=3219368504 &L[1]=3219368460 &pi[1]=3219368440
&f[2]=3219368532 &d[2]=3219368512 &L[2]=3219368472 &pi[2]=3219368444
```

MNI

104

2011-2012

7 Fonctions en C

Langage C

7 Fonctions en C

7.1 Généralités

En C, les fonctions sont **indispensables** :

⇒ la fonction principale `main` existe obligatoirement.

Une fonction sert à :

- rendre un programme plus lisible,
- factoriser des opérations répétitives.

Une fonction en C est proche d'une fonction mathématique :

- elle admet une **liste d'arguments formels**,
- elle renvoie une valeur,

le type de la fonction correspond au type de la **valeur de retour**.

Distinguer :

- arguments **formels** ou **muets** (*dummy*) dans la définition de la fonction,
- arguments **effectifs** dans l'appelant.

MNI

106

2011-2012

----- Processeur 64 bits AMD -----

Tailles en octets via sizeof :

```
char = 1 short int = 2 int = 4 long int = 8
```

Adresses converties en unsigned long int :

```
c =140735477551008 s =140735477550992 i =140735477550976 l =140735477550944
&c[0]=140735477551008 &s[0]=140735477550992 &i[0]=140735477550976 &l[0]=140735477550944
&c[1]=140735477551009 &s[1]=140735477550994 &i[1]=140735477550980 &l[1]=140735477550952
&c[2]=140735477551010 &s[2]=140735477550996 &i[2]=140735477550984 &l[2]=140735477550960
```

Tailles en octets via sizeof :

```
float = 4 double = 8 long double = 16 int * = 8
```

Adresses converties en unsigned long int :

```
f =140735477550928 d =140735477550896 L =140735477550848 pi =140735477550816
&f[0]=140735477550928 &d[0]=140735477550896 &L[0]=140735477550848 &pi[0]=140735477550816
&f[1]=140735477550932 &d[1]=140735477550904 &L[1]=140735477550864 &pi[1]=140735477550824
&f[2]=140735477550936 &d[2]=140735477550912 &L[2]=140735477550880 &pi[2]=140735477550832
```

MNI

105

2011-2012

7 Fonctions en C

Langage C

7.2 Définition d'une fonction

7.2 Définition d'une fonction

Toute fonction possède un entête et un corps et est de la forme :

```
type_retour nom_fonction(type1 arg1, type2 arg2, ...)
{
    /* declaration de variables locales */

    /* liste d'instructions */

    /* instruction de retour avant de quitter la fonction */
}
```

Remarque importante : le C n'autorise pas l'imbrication des fonctions !

⇒ la définition d'une fonction doit être placée en dehors de celle de toute autre fonction.

MNI

107

2011-2012

L'entête d'une fonction est de la forme :

```
type_retour nom_fonction(type1 arg1, ..., typen argn)
```

où :

- **type_retour** est le **type de la valeur de retour de la fonction**,
- **nom_fonction** est l'**identifiant de la fonction**,
- **arg1, ..., argn** sont les **arguments formels** (ou **muets**) de la fonction de types respectifs **type1, ..., typen**.

Le **corps** d'une fonction est placé entre **{** et **}** :

- il commence par la déclaration de **variables locales à la fonction**,
- il se poursuit par une liste d'instructions propres à la fonction,
- il se termine une **instruction de retour à la fonction appelante** :

```
return expression_de_retour ;
```

 où le type de **expression_de_retour** définit celui de la fonction.

Exemple d'une fonction de nom `som`, de type `int`, à un argument de type `const int` et possédant deux variables locales `i` et `s` de type `int` :

```
int som(const int p) /* entete de la fonction */
{
    /* corps de la fonction */
    /* somme des p premiers entiers */
    int i , s; /* declaration des variables locales */

    s = 0; /* liste d'instructions */
    for (i = 0; i <= p; i++){
        s += i;
    }

    return s ; /* instruction de retour */
}
```

7.2.1 Exemples de fonctions renvoyant une valeur

Exemple d'une fonction de nom `fonc`, de type `float`, à un argument de type `int` et ne possédant pas de variable locale :

```
float fonc(int x) /* entete de la fonction */
{
    /* corps de la fonction */
    return x/2.f ; /* instruction de retour */
}
```

Remarque importante : le corps d'une fonction doit au moins contenir une instruction de retour

7.2.2 Exemple d'une fonction sans retour

Une fonction sans retour est une fonction à **effet de bord** (*side effect*)
 ⇒ de type **void**

```
void som(const int p) /* entete de la fonction */
{
    /* corps de la fonction */
    /* somme des p premiers entiers */
    int i , s; /* variables locales */

    s = 0;
    for (i = 0; i <= p; i++){
        s += i;
    }
    /* l'affichage est un effet de bord: */
    printf("Somme des %d premiers entiers: %d\n", p, s) ;

    return ; /* aucune valeur rendue */
}
```

7.2.3 Exemple d'une fonction sans argument

Une fonction sans argument est une fonction dont l'argument est de type **void** :

```
int main(void) /* entete de la fonction */
{
    /* corps de la fonction */

    return 0 ; /* valeur de retour */
}
```

7.3 Appel d'une fonction

L'appel d'une fonction se fait au moyen de l'expression :

```
nom_fonction(paral, ..., paran)
```

où le nombre, l'ordre (et le type, cf. conversions implicites) des paramètres effectifs : **paral**, ..., doivent correspondre à ceux des arguments formels : **arg1**, ..., donnés dans l'entête de la fonction.

7.3.1 Appel d'une fonction avec retour

Si on exploite la valeur de retour, l'appel se fait au moyen de l'instruction :

```
var = nom_fonction_avec_retour(paral, ..., paran) ;
```

où **var** est une variable du même type que la valeur de retour de la fonction.

7.3.2 Appel d'une fonction sans retour

L'appel se fait au moyen de l'instruction :

```
nom_fonction_sans_retour(paral, ..., paran) ;
```

7.3.3 Appel d'une fonction sans argument

Il faut garder les parenthèses suivant le nom de la fonction.

Fonction avec retour : l'appel se fait, en général, au moyen de l'instruction :

```
var = nom_fonction_sans_argument () ;
```

où **var** est une variable du même type que la valeur de retour de la fonction.

Fonction sans retour : l'appel se fait au moyen de l'instruction :

```
nom_fonction_sans_argument () ;
```

7.4 Déclaration d'une fonction

Une fonction doit être déclarée (et/ou définie) avant d'être appelée.

La déclaration permet au compilateur de :

- vérifier la concordance entre le nombre d'arguments dans l'appel et dans la définition,
- de mettre en place des conversions implicites (pour les arguments et la valeur de retour) entre l'appel et la définition.

7.4.1 Déclaration au moyen d'un prototype (méthode conseillée)

Le prototype d'une fonction définit son type, son nom, ainsi que le nombre et le type de ses arguments :

```
type_retour nom_fonction (type1 arg1, ..., typen argn) ;
```

⇒ prototype = entête + ;

Le prototype doit être placé **avant** la définition de la fonction et

(**par ordre croissant de préférence**) :

- au sein d'une fonction ⇒ visible uniquement dans cette fonction
- au début du fichier (en dehors de toute fonction) (**conseillé pour les débutants**)
⇒ visible dans tout le fichier (déclaration globale)
- dans un fichier d'entête (ou « include »), d'extension .h, inclus par
#include "fichier.h"
⇒ visible dans tous les fichiers qui incluent ce fichier .h

La définition peut être placée n'importe où **après** le prototype et **en dehors** de toute autre fonction.

MNI 116 2011-2012

7 Fonctions en C Langage C 7.4 Déclaration d'une fonction

7.4.3 Déclaration au moyen de la définition (méthode déconseillée)

La définition tient lieu de déclaration ⇒ définir la fonction avant de l'appeler.

Problème :

cette méthode est d'autant plus limitée que le nombre de fonctions augmente ; une fonction peut en appeler d'autres qui en appellent d'autres, etc.

MNI 118 2011-2012

7.4.2 Exemple de déclaration et d'appel d'une fonction

La fonction principale est la fonction appelante :

```
#include <stdio.h>      /* pour les entrées/sorties */
#include <stdlib.h>     /* par exemple pour exit */

int som(const int p) ; /* déclaration de la fonction somme */

int main(void) {      /* fonction principale (sans param.) */
    int s ;

    s = som(5) ;      /* appel de la fonction somme */

    printf("somme entiers de 1 à 5 = %d\n", s);
    printf("somme entiers de 1 à 9 = %d\n", som(9)); /*autre appel*/

    exit(0) ;        /* renvoie à unix un status 0 (OK) */
}
```

MNI 117 2011-2012

7 Fonctions en C Langage C 7.5 Quelques fonctions (standards) du C

7.5 Quelques fonctions (standards) du C

7.5.1 La fonction principale : main

La fonction **main** est la fonction principale d'un programme C :

celle par où le programme débute

⇒ elle existe obligatoirement

L'entête de la fonction principale est généralement de la forme :

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

– **valeur de retour** du type **int**

⇒ le corps de la fonction doit se terminer par une instruction de retour :

```
return 0;          qui renvoie au shell la valeur 0
```

ou bien :

```
exit(EXIT_SUCCESS); qui renvoie au shell la valeur EXIT_SUCCESS (=0)
```

– **arguments** : reçus de l'interpréteur de commande au lancement de l'exécution.

Utilisation sans argument ⇒ **int main(void)**

MNI 119 2011-2012

7.5.2 La fonction `exit`

Fonction de la bibliothèque standard.

La fonction `exit` met fin à un processus (l'exécution du programme).

Son prototype est dans le fichier : `stdlib.h`

⇒ directive préprocesseur (compilation) : `#include <stdlib.h>`

Prototype : `void exit (int status) ;`

- valeur de retour du type `void`
- argument `status` de type `int` et dont les valeurs sont généralement :
 - `0` ou `EXIT_SUCCESS` pour signaler que le programme se termine normalement
 - `1` ou `EXIT_FAILURE` pour signaler un problème

Remarque : Le fichier `stdlib.h` contient aussi les constantes prédéfinies :

`EXIT_SUCCESS`, `EXIT_FAILURE`, `NULL`, etc...

7.5.3 Les fonctions `printf` et `scanf`

Les fonctions `printf` et `scanf` sont des fonctions d'entrée-sortie standard.

Leur prototype est dans le fichier : `stdio.h`

⇒ directive préprocesseur (compilation) : `#include <stdio.h>`

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

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

- valeur de retour du type `int`
 - `printf` : le nombre de caractères écrits si tout se passe bien ou un nombre négatif en cas d'erreur.
 - `scanf` : le nombre de variables effectivement lues si tout se passe bien ou un nombre négatif en cas d'erreur.
- nombre variable d'arguments : `...`, dont au moins une chaîne de caractères

Remarque : si on ne s'intéresse pas aux valeurs de retour de ces fonctions, elles peuvent être appelées comme des fonctions sans retour.

7.5.4 Les fonctions mathématiques (`pow`, `fabs`, ...)

Ce sont les fonctions trigonométriques, exponentielles, logarithmiques, etc...

⇒ **importance capitale pour les applications numériques.**

Les prototypes sont contenus :

- **en C89**, dans le fichier de librairie standard `math.h`
 - ⇒ directive préprocesseur (compilation) : `#include <math.h>`
- **en C99**, dans le fichier de librairie standard `tgmath.h`
 - ⇒ directive préprocesseur (compilation) : `#include <tgmath.h>`
 - (`tgmath.h` permet d'utiliser le type `complex` absent de `math.h`)

Remarque importante : pour pouvoir utiliser les fonctions de la bibliothèque mathématique il faut utiliser l'option `-lm` de `gcc` (édition des liens).

Exemple de la fonction puissance : `pow`

En C, il n'y a pas d'opérateur puissance mais une fonction puissance :

$$x^y \Rightarrow \text{pow}(x, y)$$

Prototype : `double pow (double x, double y)` ;

Cette fonction peut être utilisée avec des paramètres de type entier ou float

⇒ le processeur se charge de convertir les paramètres en double

⇒ le résultat est aussi un double

Exemple de la fonction valeur absolue : `fabs`

Fonction renvoyant la valeur absolue d'un nombre réel :

$$|x| \Rightarrow \text{fabs}(x)$$

Prototype : `double fabs (double x)` ;

Remarque importante : ne pas confondre `fabs` et `abs`

Prototype de `abs` : `int abs (int i)` ;

7.5.5 Exemple d'un programme utilisant math.h

```

/* programme math_exemples.c */
#include <stdio.h> /* entrees-sorties */
#include <stdlib.h> /* exit et EXIT_SUCCESS */
#include <math.h> /* fonctions mathematiques */

int main(void) {
    int i=2;
    double x=-0.5, y;

    y = abs(x); /* attention: conversion en entier */
    printf("Valeur absolue entiere de %g: %g\n", x, y);

    y = fabs(x);
    printf("Valeur absolue reelle de %g: %g\n", x, y);
}

```

```

y = pow(x, (double) i);
printf(" %g puissance %d = %g\n", x, i, y);

exit(EXIT_SUCCESS);
}

```

Compilation avec :

```
gcc-mni math_exemples.c -lm -o math_exemples.x
```

Résultat de l'exécution :

```

Valeur absolue de -0.5: 0
Valeur absolue de -0.5: 0.5
-0.5 puissance 2 = 0.25

```

7.5.6 Liste des fonctions mathématiques standards

C89	C99 avec <code>tgmath.h</code>	remarques
l/ll/abs(n) (i) f abs/f/l(a) (r)	fabs(a) (a) cabs(c) (z) C99 + fabs	abs pour les entiers C99 + tgmath.h complex.h
sqrt/f/l(a) (r) pow /f/l(a,b) (r)	csqrt(a) (z) sqrt cpow/f/l(a,b) (z) pow	
exp/f/l(a) (r)	cexp/f/l(a) (z) exp	
log/f/l(a) (r)	clog/f/l(a) (z) log	
log10/f/l(a)		
	cimag/f/l(a) (z) conj/f/l(a) (z) conj	C99 + complex.h C99 + complex.h
ceil/f/l(a)	ceil	résultat flottant
floor/f/l(a)	floor	résultat flottant
n%p		opérateur
	copysign/f/l(a,b)	

C89	C99 avec <code>tgmath.h</code>	remarques
cos/f/l(a) (r)	ccos/f/l(a) (z) cos	
cosh/f/l(a) (r)	ccosh(a) (z) cosh	
acos/f/l(a) (r)	caacos/f/l(a) (z) acos	C99 + tgmath.h
sin/f/l(a) (r)	csin/f/l(a) (z) sin	
sinh/f/l(a) (r)	csinh/f/l(a) (z) sin	C99 + tgmath.h
asin/f/l(a) (r)	casin/f/l(a) (z) asin	
tan/f/l(a) (r)	ctan/f/l(a) (z) tan	
tanh/f/l(a) (r)	ctanh/f/l(a) (z) tanh	C99 + tgmath.h
atan/f/l(a) (r)	catan/f/l(a) (z) atan	
atan2/f/l(a,b)	atan2	

Remarque : accès via le shell à la documentation en ligne au moyen de la commande :

```
man 3 nom_fonction
```


7.6 La portée des variables

La portée (*scope* en anglais) d'une variable correspond à la portion de programme sur laquelle la variable est définie.

Pour un programme donné, on distingue deux cas :

- **variable globale** : variable dont la portée s'étend à l'ensemble du programme,
- **variable locale** : variable dont la portée est limitée à une portion du programme (une fonction par exemple, voire un sous-bloc d'une fonctions).

Conseils pratiques :

- les variables globales doivent rester des exceptions
⇒ **privilégier l'emploi de variables locales.**
- garder les mêmes notations pour les variables globales et locales (lettres minuscules)
⇒ réserver les majuscules aux constantes préprocesseur

MNI 128 2011-2012

7 Fonctions en C Langage C 7.6 La portée des variables

7.6.3 Exemple d'un programme utilisant une variable globale

```

/* programme global_exemple.c */
#include <stdio.h>          /* pour les entrées/sorties */
#include <stdlib.h>        /* par exemple pour exit */

int n=5;                   /* declaration variable globale n=5 */
int somme(void) ;          /* déclaration de la fonction somme */

int main(void){           /* fonction principale */
    int s ;                /* variable locale a main */
    s = somme() ;          /* appel fonction somme sans argument */
    printf("somme de 1 a %d = %d\n", n, s) ;
    n++;                  /* incrementation de la variable n */
    s = somme() ;          /* appel de la fonction somme */
    printf("somme de 1 a %d = %d\n", n, s) ;
    exit(0) ;             /* renvoie à unix un status 0 (OK) */
}

```

MNI 130 2011-2012

7.6.1 Variables globales

Une variable globale est déclarée avant toute fonction (conseillé)

⇒ portée : ensemble du programme

⇒ **variable dite permanente ou statique** : occupe la même zone mémoire pendant toute l'exécution du programme.

Remarque : une variable globale peut être déclarée **entre** deux fonctions (**déconseillé**)

⇒ portée : tout le programme **suivant** la déclaration de la variable

7.6.2 Variables locales

Une variable locale à un bloc est déclarée dans ce bloc

⇒ portée : réduite au bloc dans laquelle elle est déclarée

⇒ **variable dite temporaire ou automatique** : la zone mémoire occupée par la variable est libérée à la sortie du bloc.

Remarque : le **C99** autorise les **déclarations tardives**

⇒ la portée peut être restreinte à un sous-bloc d'une fonction

MNI 129 2011-2012

7 Fonctions en C Langage C 7.6 La portée des variables

```

int somme(void)            /* définition de la fonction somme */
{
    int i , sum ;        /* variables locales a somme */
    for (i = 0, sum = 0 ; i <= n ; i++) /* n: variable globale ! */
    {
        sum += i ;
    }
    return sum ;         /* valeur rendue par la fonction */
}

```

Résultats :

somme de 1 a 5 = 15

somme de 1 a 6 = 21

MNI 131 2011-2012

7.7 Passage de paramètres dans une fonction

Par passage de paramètre on entend la transmission de la valeur d'une variable de la fonction appelante vers la fonction appelée et vice-versa.

Rappel :

L'appel d'une fonction se fait au moyen de l'expression :

```
nom_fonction(paral, ..., paran)
```

où le nombre, l'ordre (et le type) des paramètres : `paral`, ..., doivent correspondre à ceux des arguments : `arg1`, ..., donnés dans l'entête de la fonction.

Important : on utilise la terminologie suivante

- Les **paramètres effectifs** sont des variables de la **fonction appelante**.
- Les **paramètres formels (ou muets)** d'une fonction sont ceux qui figurent dans **l'entête de la fonction**.

```
printf("\tfin echange : %d %d \n", a, b);
printf("\tadresses fin echange : %p %p \n", &a, &b);
return;
}

int main(void)
{
    int n = 1, p = 5;
    printf("avant appel : n=%d p=%d \n", n, p);
    printf("adresses avant appel : %p %p \n", &n, &p);
    echange(n,p); /* appel avec les parametres effectifs */
    printf("apres appel : n=%d p=%d \n", n, p);
    printf("adresses après appel : %p %p \n", &n, &p);
    exit(0) ;
}
```

7.7.1 Exemple de passage par valeur : le faux échange

```
/* programme faux-echange.c */
/* Fonctions : passage des arguments par valeur */
/* => pas de modification en retour */
#include<stdio.h>
#include<stdlib.h>
void echange(int a, int b) ; /* prototype */
void echange(int a, int b) /* definition */
{ /* a et b:      ^      ^      parametres formels (muets) */
    int c; /* variable locale à la fonction */
    printf("\tdebut echange : %d %d \n", a, b);
    printf("\tadresses debut echange : %p %p \n", &a, &b);
    c = a;
    a = b;
    b = c;
}
```

Les paramètres muets, `a` et `b`, de la fonction `echange` stockent **la valeur** des paramètres effectifs, `n` et `p`, de la fonction appelante :

⇒ `a` et `b` sont une copie locale de la valeur de `n` et `p`, respectivement

⇒ la copie disparaît au retour dans la fonction appelante

⇒ les variables de la fonction appelante **ne sont pas modifiées** par la fonction appelée

⇒ on parle de **passage par copie de valeur**

avant appel : n=1 p=5

adresses avant appel : 0xbfd04174 0xbfd04170

debut echange : 1 5

adresses debut echange : 0xbfd04140 0xbfd04144

fin echange : 5 1

adresses fin echange : 0xbfd04140 0xbfd04144

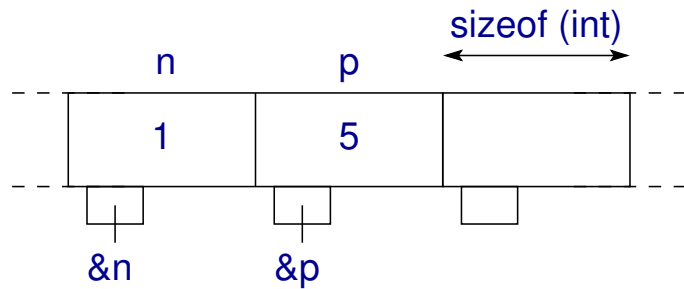
apres appel : n=1 p=5

adresses après appel : 0xbfd04174 0xbfd04170

7.7.2 Le faux échange : visualisation de la RAM à l'exécution

Dans la fonction `main` avant l'appel à la fonction `echange` :

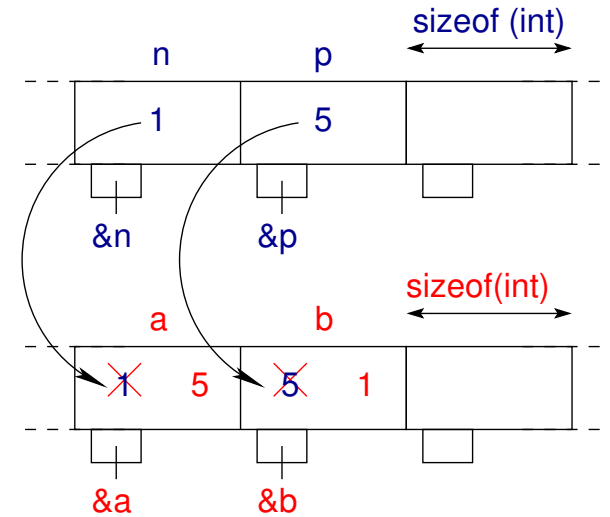
```
int n=1; int p=5;
```



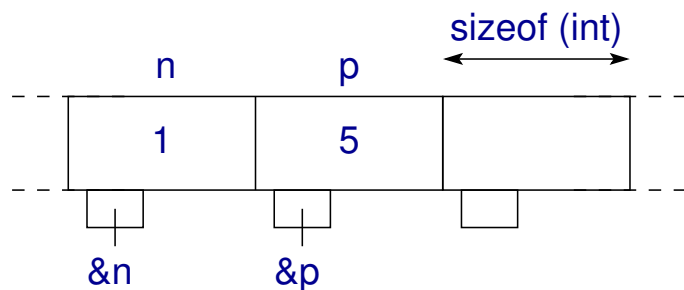
Attention : pour simplifier, chaque segment représenté a la taille d'un entier.

A l'appel de la fonction `echange` : passage par copie de valeur

⇒ échange de `a` et `b`



Au retour à la fonction `main` les valeurs de `n` et `p` restent inchangées :



7.7.3 Exemple de passage par adresse : le vrai échange

```
/* programme echange.c */
/* Fonctions : passage des adresses en arguments (par valeur)
/* => modification en retour */
#include<stdio.h>
#include<stdlib.h>
void echange(int *ptrb, int *ptrb); /* prototype */
void echange(int *ptrb, int *ptrb){ /* definition */
/* ptrb et ptrb ^ ^ pointeurs sur des entiers */
int c; /* variable locale à la fonction (pas pointeur) */
printf("\tdebut echange : %d %d \n", *ptrb, *ptrb);
printf("\tadresses debut echange : %p %p \n", ptrb, ptrb);
c = *ptrb;
*ptrb = *ptrb;
*ptrb = c;
```

```

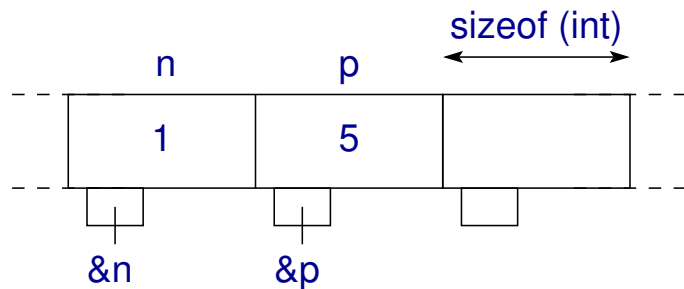
printf("\tfin échange : %d %d \n", *ptra, *ptrb);
printf("\tadresses fin échange : %p %p \n", ptra, ptrb);
return;
}
int main(void)
{
    int n = 1, p = 5;
    printf("avant appel : n=%d p=%d \n", n, p);
    printf("adresses avant appel : %p %p \n", &n, &p);
    échange(&n, &p); /* appel avec les adresses */
    printf("après appel : n=%d p=%d \n", n, p);
    printf("adresses après appel : %p %p \n", &n, &p);
    exit(0);
}

```

7.7.4 Le vrai échange : visualisation de la RAM à l'exécution

Dans la fonction `main` avant l'appel à la fonction `échange` :

```
int n=1; int p=5;
```



Attention : pour simplifier, chaque segment représenté a la taille d'un entier ou d'un pointeur sur un entier.

Les paramètres muets, `ptra` et `ptrb`, de la fonction `échange` stockent l'adresse des paramètres effectifs, `n` et `p`, de la fonction appelante :

⇒ `ptra` et `ptrb` sont des **pointeurs** sur `n` et `p`, respectivement

⇒ la fonction appelée travaille sur la même zone mémoire que la fonction appelante

⇒ la fonction appelée **modifie** (indirectement) la valeur de **plusieurs paramètres** de la fonction appelante

⇒ on parle de **passage par copie d'adresse**

avant appel : n=1 p=5

adresses avant appel : 0xbffbf524 0xbffbf520

debut échange : 1 5

adresses debut échange : 0xbffbf524 0xbffbf520

fin échange : 5 1

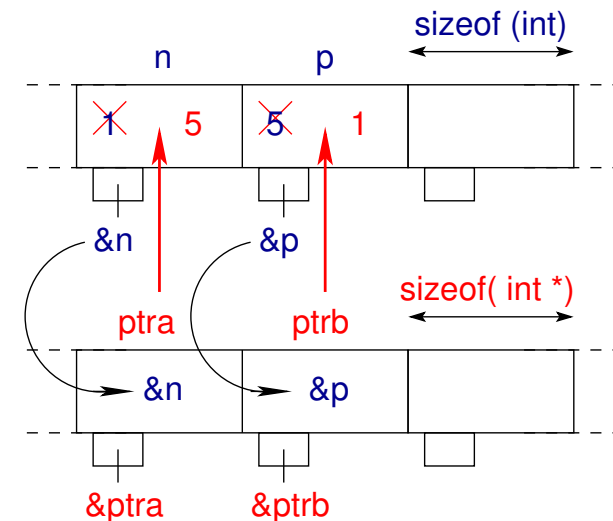
adresses fin échange : 0xbffbf524 0xbffbf520

après appel : n=5 p=1

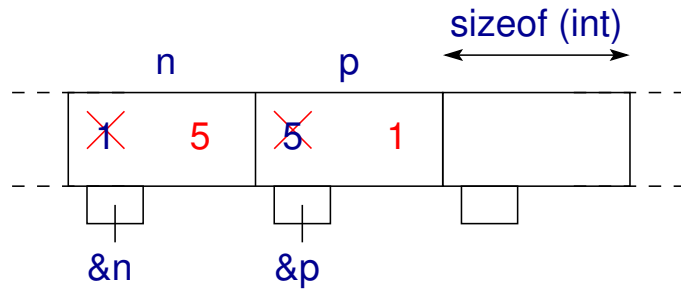
adresses après appel : 0xbffbf524 0xbffbf520

A l'appel de la fonction `échange` : passage par copie d'adresse

⇒ échange de ***ptra** et ***ptrb**



Au retour à la fonction `main` les valeurs de `n` et `p` ont bien été échangées :



7.8 Retour sur printf/scanf

`printf` ne modifie pas les variables qu'il reçoit en argument

⇒ **passage par copie de valeur**

Dans la fonction appelante :

```
double x = 0.2 ; float y = 3.4f ; int n = 10 ;
printf("x=%g, y=%g, n=%d\n", x, y, n);
```

(conversions implicites possibles)

`scanf` modifie la valeur des variables qu'il reçoit à partir du deuxième argument

⇒ il est possible de modifier la valeur de **plusieurs** variables

⇒ **passage par copie de valeur** du premier argument

⇒ **passage par copie d'adresse** des arguments qui suivent

Dans la fonction appelante :

```
double x ; float y ; int n ;
scanf("%lg %g %d", &x, &y, &n);
```

(attention : le pointeur **doit** être du même type que la variable pointée)

7.7.5 Bilan sur le passage de paramètres

Si l'on souhaite **modifier une seule variable** de la fonction appelante :

⇒ privilégier le **passage par copie de valeur** des variables que l'on ne souhaite pas modifier

(conversions implicites possibles)

⇒ utiliser la **valeur de retour** de la fonction pour modifier la variable concernée

Si l'on souhaite **modifier deux variables ou plus** de la fonction appelante :

⇒ privilégier le **passage par copie d'adresse** des variables à modifier (pointeurs)
(chaque pointeur **doit** être du même type que la variable pointée)

⇒ passage par valeur des variables que l'on ne souhaite pas modifier

⇒ fonction avec ou sans retour selon les besoins

Remarque importante : dans le passage par copie d'adresse il faut respecter exactement le type (éviter la conversion de type de pointeur)

7.9 Complément sur les fonctions : la récursivité

Fonction récursive = fonction qui s'appelle elle-même

7.9.1 Exemple de fonction récursive : factorielle

```
/* programme fct_rekurs.c */
int fact(int n){ /* calcul récursif de factorielle */
/* attention aux dépassements de capacité non testés en entier */
int factorielle;
if ( n > 0 ){
    factorielle = n * fact(n-1); /* provoque un autre appel à fact */
}
else {
    factorielle = 1 ; /* fin de la recursion */
}
return factorielle;
}
```

8 Tableaux

8.1 Définition et usage

Un **tableau** est un ensemble "rectangulaire" d'éléments :

- de même type,
- repérés au moyen d'**indices entiers**,
- stockés en mémoire à des **adresses contiguës**.

L'ensemble de ces éléments est identifié par un **identifiant unique** :

le nom du tableau.

Les tableaux sont utilisés lorsque l'on manipule plusieurs objets de même type,

ayant un lien entre eux :

- les différentes valeurs d'une fonction (vecteur de N points),
- les coordonnées de plusieurs points
(matrice de N points $\times 2$ (ou 3) coordonnées)
- etc...

MNI 148 2011-2012
8 Tableaux Langage C 8.1 Définition et usage

Résultat à l'exécution :

```
element 1 du tableau = tabld[0] = 0.5
element 2 du tableau = tabld[1] = 1.5
element 3 du tableau = tabld[2] = 2.5
```

Caractéristiques de ce programme :

- affectation globale du tableau dès sa déclaration (initialisation)
- utilisation d'une boucle pour l'impression des éléments du tableau

MNI 150 2011-2012

8.1.1 Exemples de programmes élémentaires utilisant des tableaux 1D

```
/* fichier tableaux-elem1d_1.c */
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int i;

    // declaration + affectation d'un tableau de 3 reels (double)
    double tabld[3] = {0.5, 1.5, 2.5}; // taille fixe de 3
    //          ^ un seul indice: tableau 1D
    // impression du tableau a l'aide d'une boucle for:
    for (i=0; i<3; i++){ // attention: indice i varie de 0 a 2
        printf("element %d du tableau = tabld[%d] = %g\n", i+1, i, tabld[i]);
    }

    exit(EXIT_SUCCESS);
}
```

MNI 149 2011-2012
8 Tableaux Langage C 8.1 Définition et usage

```
/* fichier tableaux-elem1d_2.c */
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int i;

    // declaration d'un tableau de 3 reels (double):
    double tabld[3]; // taille fixe de 3
    //          ^ un seul indice: tableau 1D
    // affectation du tableau a l'aide d'une boucle for:
    for (i=0; i<3; i++){ // attention: indice i varie de 0 a 2
        tabld[i] = (double) i + 0.5;
    }

    // impression du tableau a l'aide d'une boucle for:
    for (i=0; i<3; i++){ // attention: indice i varie de 0 a 2
        printf("element %d du tableau = tabld[%d] = %g\n", i+1, i, tabld[i]);
    }
    exit(EXIT_SUCCESS);
}
```

MNI 151 2011-2012

8.1.2 Exemples de programmes élémentaires utilisant des tableaux 2D

Résultat à l'exécution :

```

element 1 du tableau = tab1d[0] = 0.5
element 2 du tableau = tab1d[1] = 1.5
element 3 du tableau = tab1d[2] = 2.5

```

Caractéristiques de ce programme :

- utilisation d'une boucle pour l'affectation du tableau
- utilisation d'une boucle pour l'impression des éléments du tableau

```

/* fichier tableaux-elem2d_0.c */
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int i, j;
    // declaration + affectation d'un tableau de 6 entiers
    int tab2d[2][3] = {{1,2,3},{4,5,6}}; // taille fixe de 6
    //      ^      ^ deux indices: tableau 2D
    // impression du tableau a l'aide de deux boucles for:
    for (i=0; i<2; i++){ // attention: indice i varie de 0 a 1
        for (j=0; j<3; j++){ // attention: indice j varie de 0 a 2
            printf("tab2d[%d][%d] = %d\n", i, j, tab2d[i][j]);
        }
    }
    exit(EXIT_SUCCESS);
}

```

Résultat à l'exécution :

```

tab2d[0][0] = 1
tab2d[0][1] = 2
tab2d[0][2] = 3
tab2d[1][0] = 4
tab2d[1][1] = 5
tab2d[1][2] = 6

```

```

/* fichier tableaux-elem2d_1.c */
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int i, j;
    // declaration d'un tableau de 6 entiers
    int tab2d[2][3]; // taille fixe de 6
    //      ^      ^ deux indices: tableau 2D
    // affectation du tableau
    for (i=0; i<2; i++){ // attention: indice i varie de 0 a 1
        for (j=0; j<3; j++){ // attention: indice j varie de 0 a 2
            tab2d[i][j] = 3*i + j + 1;
        }
    }
    // impression du tableau sous forme de matrice:
    for (i=0; i<2; i++){ // attention: indice i varie de 0 a 1
        for (j=0; j<3; j++){ // attention: indice j varie de 0 a 2

```

```

    printf(" %d ", tab2d[i][j]);
}
printf("\n");
}
exit(EXIT_SUCCESS);
}

```

Résultat à l'exécution :

```

1  2  3
4  5  6

```

8.2 Tableaux de taille fixe

8.2.1 Déclaration d'un tableau de taille fixe

- déclaration d'un tableau à **1 indice (vecteur)** de 3 **réels (double)** :
`double tab1d[3] ;`
- déclaration d'un tableau à **2 indices (matrice)** de 6 **entiers** :
`int tab2d[2][3] ;`
- etc ...

La **dimension** d'un tableau correspond au **nombre d'indices** (ou de "directions")

⇒ un tableau à 1 indice est un tableau 1D (un vecteur)

⇒ un tableau à 2 indices est un tableau 2D (une matrice)

La **taille** d'un tableau correspond à l'espace occupé par les variables stockées dans les éléments du tableau

⇒ un tableau 1D de 3 entiers a une taille de $3 * \text{sizeof}(\text{int})$

⇒ un tableau 2D de 6 doubles a une taille de $6 * \text{sizeof}(\text{double})$

Cas général :

déclaration d'un tableau de taille fixe à **n indices** de $n_1 * n_2 * \dots * n_n$ **variables** :

```
type_tab nom_tab[n1][n2]...[nn] ;
```

⇒ dimension : n (on a n "directions")

⇒ taille : $n_1 * n_2 * \dots * n_n * \text{sizeof}(\text{type_tab})$

- **type_tab** est le **type des variables stockées dans les éléments du tableau**,
- **nom_tab** est l'**identifiant du tableau**,
- n_i est une **constante** entière positive correspondant au **nombre d'éléments dans la i -ème direction**.

Attention : la dimension d'un tableau peut faire référence :

- au nombre d'indices (ou directions) ⇒ **convention adoptée ici**.
- à la valeur d'un indice donné. Par exemple : 3 dans le cas de

```
double tab1d[3] ;
```

8.2.2 Indexation et référence à un élément d'un tableau

Pour un tableau à 1 indice, de **n éléments**, l'**indice** varie de **0 à n-1**.

Exemple : `tab1d[2]` est le **troisième** élément du tableau.

Exemples de références à des éléments de tableaux :

Tableau 1D de doubles : `x = tab1d[2]` ; où **x** est un double

⇒ on stocke le 3ème élément de `tab1d` dans **x**

Tableau 2D d'entiers : `i = tab2d[1][0]` ; où **i** est un entier

⇒ on stocke le 4ème élément de `tab2d` dans **i**

Attention : pas d'opérateur séquentiel `<<, >>` ⇒ `[2, 0]` est interprété comme `[0]`

Attention : généralement, aucun contrôle sur la valeur de l'indice n'est effectué par le compilateur

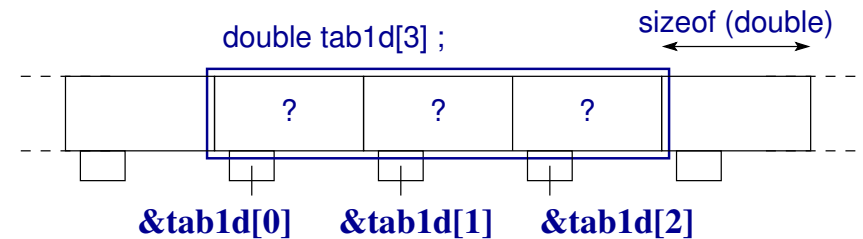
⇒ en cas de **débordement d'indice** il y a un risque important d'erreur de

8.2.3 Déclaration d'un tableau 1D : visualisation de la RAM

Déclarer un tableau \Rightarrow **réserver une zone mémoire** pour stocker l'ensemble de ses éléments :

- la **taille** de cette zone dépend du **type** du tableau et du **nombre** de ses éléments,
- l'**emplacement** des éléments du tableau en mémoire correspond à des **adresses contiguës**.

Exemple de déclaration d'un tableau à 1 dimension : `double tab1d[3] ;`



segmentation (voir plus loin)

8.2.4 Affectation d'un tableau

On distingue deux manières d'affecter un tableau de taille fixe :

A la déclaration : (initialisation)

- Pour un tableau à une dimension (vecteur) :

```
double tab1d[3] = {0.5, 1.5, 2.5};
```

produit le tableau `tab1d` tel que `tab1d[0]=0.5`, `tab1d[1]=1.5` et `tab1d[2]=2.5`.

- Pour un tableau à deux dimensions (matrice) :

```
int tab2d[2][3] = {{1, 2, 3}, {4, 5, 6}} ;
```

produit le tableau `tab2d` tel que `tab2d[0][0]=1`, `tab2d[0][1]=2`, `tab2d[1][0]=4`, etc ...

Remarque : tableau 2d = tableau de tableaux

\Rightarrow cas de `tab2d[2][3]` : 1 tableau de 2 tableaux de 3 éléments de type `int`

\Rightarrow pas de véritable notion de tableau multidimensionnel en C.

Hors de la déclaration : (**méthode la plus courante**)

il est alors impossible de préciser les valeurs d'un tableau en une seule instruction
 \Rightarrow recours aux **boucles** en général

- Pour le tableau `tab1d` précédent :

```
double tab1d[3] ;
```

```
for (i=0; i<3; i++) { /* indice variant de 0 a 2 */
    tab1d[i] = i + 0.5 ;
}
```

- Pour le tableau `tab2d` précédent :

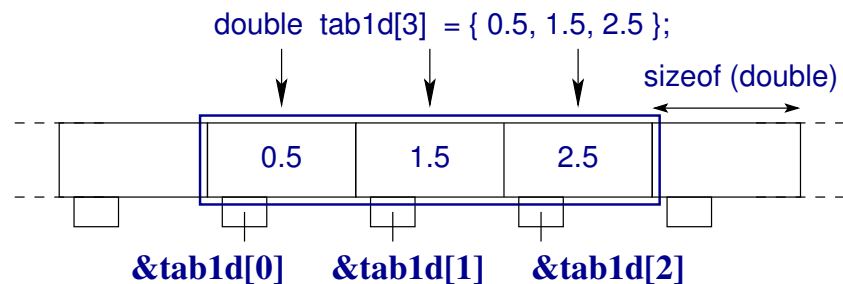
```
int tab2d[2][3] ;
```

```
for (i=0; i<2; i++) { /* indice lent */
    for (j=0; j<3; j++) { /* indice rapide */
        tab2d[i][j] = 3*i + j + 1 ;
    }
}
```

8.2.5 Affectation d'un tableau 1D : visualisation de la RAM

Exemple d'initialisation d'un tableau :

```
double tab1d[3] = {0.5, 1.5, 2.5};
```

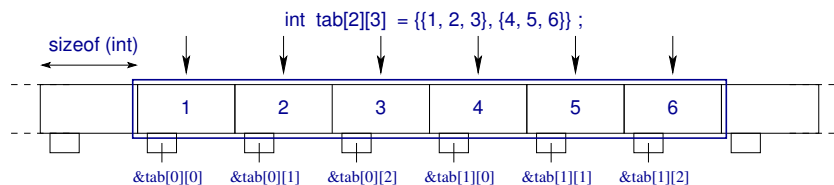


Attention : en cas de **débordement d'indice** il y a un risque important d'erreur de segmentation

8.2.6 Affectation d'un tableau 2D : visualisation de la RAM

Exemple d'initialisation d'un tableau 2D :

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



Remarque importante :

Dans le cas de tableaux à plusieurs dimensions

l'indice des éléments contigus est **l'indice le plus à droite**.

8.2.7 Ordre des éléments de tableaux 2D en C

Dans le cas de tableaux à plusieurs dimensions

l'indice qui défile le plus vite est celui des **éléments contigus**

⇒ c'est donc **l'indice le plus à droite**.

Ainsi, dans le cas d'un tableau 2D :

- 1er indice = "lignes" de la matrice
- 2ème indice (le plus rapide) = "colonnes" de la matrice

Conséquence : la boucle la plus interne est sur l'indice rapide

⇒ augmente la vitesse de calcul

(un bon compilateur optimise automatiquement le code de cette manière)

8.3 Inconvénients des tableaux de taille fixe

Très fréquemment, on est amené à exécuter un même programme sur un ensemble d'éléments dont **le nombre varie d'une exécution à l'autre**

(exemple : la résolution avec laquelle on examine une fonction)

⇒ l'utilisation des tableaux de taille fixe (tel que ci-dessus) nécessite une modification de nombreuses lignes de code (risque d'erreurs)

On étudiera trois manières de remédier à cet inconvénient :

- l'utilisation d'une **directive préprocesseur**
- l'utilisation de **tableaux automatiques**,
- **l'allocation dynamique** (prochain chapitre)

8.3.1 Directive préprocesseur (ancienne méthode)

Utiliser une directive du préprocesseur pour **paramétrer la taille** :

```
#define N 20 (noter l'utilisation de la majuscule)
int main(void) {
    int i, j ;
    int tab2d[N][2*N+1] ;
    ...
    for (i=0; i<N; i++) {
        ...
        for (j=0; j<2*N+1; i++) {
            ...
        }
    }
}
```

Avantage : changer la taille du tableau ne nécessite que la modification de la 1ère ligne.

Inconvénient : changer la taille du tableau nécessite une recompilation.

MNI 168 2011-2012

8 Tableaux Langage C 8.3 Inconvénients des tableaux de taille fixe

```
printf("impression du tableau 2D t[i][j]=10*i+j+11\n");
printf("en faisant varier j à i fixé\n");
for (i=0; i<LIGNES; i++) { // indice lent
    for (j=0; j<COLONNES; j++) { // indice rapide
        printf("%d ", t[i][j]) ;
    }
    printf("\n") ;
}
exit(0) ;
}
```

Résultat à l'exécution :

```
impression du tableau 2d t[i][j]=10*i+j+11
en faisant varier j à i fixé
11 12 13 14 15
21 22 23 24 25
31 32 33 34 35
```

MNI 170 2011-2012

```
/* programme tableaux-lem2d_2.c */
#include <stdio.h>
#include <stdlib.h>
#define LIGNES 3 // taille dans la 1ere direction
#define COLONNES 5 // taille dans la 2eme direction

int main(void) {
    int i, j;
    // declaration d'un tableau de 15 elements:
    int t[LIGNES][COLONNES]; // taille fixe de 15

    // affectation du tableau:
    for (i=0; i<LIGNES; i++) { // indice lent
        for (j=0; j<COLONNES; j++) { // indice rapide
            t[i][j] = 10*i + j + 11 ;
        }
    }
}
```

MNI 169 2011-2012

8 Tableaux Langage C 8.3 Inconvénients des tableaux de taille fixe

8.3.2 Déclaration tardive et tableau automatique (méthode conseillée)

En **C99**, on peut profiter de la possibilité offerte par les **déclarations tardives** :

```
int n ;
scanf("%d", &n) ; /* on demande à l'utilisateur la taille */
double tab[n] ;
```

Le tableau `tab` est déclaré **APRES** une instruction \Rightarrow déclaration tardive.

Avantage :

Le code est compilé une fois pour toutes.

La valeur de `n` est modifiée à l'exécution seulement (taille "variable").

Remarque importante : Il faut affecter `n` **AVANT** de déclarer le tableau.

Le code suivant compile, mais a de grandes chances de provoquer une erreur :

```
int n ;
double tab[n] ; /* la valeur de n est indéfinie ici !!*/
```

MNI 171 2011-2012

8.3.3 Exemple de programme utilisant un tableau automatique

```

/* programme declar-tardive.c */
#include <stdio.h>
#include <stdlib.h>
//----- attention: norme C99 -----
int main(void) {
    int n; // taille du tableau 1D (vecteur)

    printf("Entrer la taille du vecteur\n");
    scanf("%d", &n); // taille fixée par l'utilisateur
    /* tableau de taille variable => decl. tardive
    on peut utiliser tab partout sous sa déclaration: */
    double tab[n];

    printf("*** Saisie des elements ***\n");
    for (int i=0; i<n; i++) { // decl. tardive de i locale au sous-bloc
        printf("tab[%d] ? ", i);
        scanf("%lg", &tab[i]);
    } // sortie du sous-bloc: la zone memoire occupee par i est liberee

```

```

printf("*** Affichage des elements ***\n");
for (int i=0; i<n; i++) { // decl. tardive de i locale au sous-bloc
    // i est une nouvelle variable sans lien avec la précédente
    printf("tab[%d] = %g\n", i, tab[i]);
}
exit(EXIT_SUCCESS);
}

```

Entrer la taille du vecteur

3

*** Saisie des elements ***

tab[0] ?

tab[1] ?

tab[2] ?

*** Affichage des elements ***

tab[0] = 2.5

tab[1] = -4.45

tab[2] = 0.123

8.4 Tableaux et pointeurs en C

8.4.1 Notions de base

- un **tableau** est un **pointeur constant sur le premier élément du tableau**,
- la **valeur** de ce pointeur est l'**adresse du premier élément du tableau**.

Exemple: float tab[9] ;

⇒ **tab** est un pointeur vers **tab[0]**

⇒ **tab** correspond à **&tab[0]**

⇒ ***tab** correspond à **tab[0]**

L'affectation globale **tab = ...** est donc **impossible**

Application :

```
scanf("%g", tab); /* equivalente a: scanf("%g", &tab[0]); */
```

⇒ conversion de **tab** en un pointeur constant vers **tab[0]**

8.4.2 Arithmétique des pointeurs

On peut faire des opérations arithmétiques (addition, soustraction) sur des pointeurs

⇒ cela revient à se déplacer dans la mémoire.

Exemple dans le cas d'un tableau: float tab[9] ;

⇒ **tab + i** correspond à **&tab[i]**

⇒ ***(tab + i)** correspond à **tab[i]**

Exemple dans le cas d'un pointeur ordinaire: float *pf ;

```
pf = &tab[3] ; /* equivalente a: pf = tab + 3 ; */
```

si **pf** correspond à **&tab[3]** ⇒ **pf++** correspond à **&tab[4]**

si **pf** correspond à **&tab[3]** ⇒ **pf--** correspond à **&tab[2]**

⇒ **pf** sous-tableau commençant au 4^e élément de **tab**

(attention : on peut revenir en arrière)

8.4.3 Retour sur l'ordre des éléments de tableaux 2D en C

```

/* programme tableaux-elem2d_3.c */
#include <stdio.h>
#include <stdlib.h>
#define LIGNES 3 // taille dans la 1ere direction
#define COLONNES 5 // taille dans la 2eme direction
int main(void) {
    int i, j;
    int t[LIGNES][COLONNES]; // taille fixe de 15
    int *k = &t[0][0]; // pointeur sur t[0][0]
    // affectation du tableau:
    for (i=0; i<LIGNES; i++) { // indice lent
        for (j=0; j<COLONNES; j++) { // indice rapide
            t[i][j] = 10*i + j + 11 ;
        }
    }
}

```

MNI

176

2011-2012

8 Tableaux

Langage C

8.4 Tableaux et pointeurs en C

Résultat à l'exécution :

```

impression du tableau 2d t[i][j]=10*i+j+11
en faisant varier j à i fixé
11 12 13 14 15
21 22 23 24 25
31 32 33 34 35
impression du tableau 2d t[i][j]=10*i+j+11
en suivant l'ordre en mémoire
11 12 13 14 15 21 22 23 24 25 31 32 33 34 35

```

⇒ l'indice **le plus à droite** varie le plus vite

MNI

178

2011-2012

```

printf("impression du tableau 2D t[i][j]=10*i+j+11\n");
printf("en faisant varier j à i fixé\n");
for (i=0; i<LIGNES; i++) { // indice lent
    for (j=0; j<COLONNES; j++) { // indice rapide
        printf("%d ", t[i][j]) ;
    }
    printf("\n") ;
}
printf("impression du tableau 2D t[i][j]=10*i+j+11\n");
printf("en suivant l'ordre en memoire\n");
for (i=0; i<LIGNES*COLONNES; i++) {
    printf("%d ", *(k+i)) ; // ou meme k[i]
}
printf("\n") ;
printf("=> l'indice le plus a droite varie le plus vite\n") ;
exit(0) ;
}

```

MNI

177

2011-2012

8 Tableaux

Langage C

8.4 Tableaux et pointeurs en C

8.4.4 Sous-tableau 1D avec un pointeur

```

/* programme sous-tab1d.c */
#include <stdio.h>
#include <stdlib.h>
#define N 10 // taille du tableau 1D
/* manipulation d'un sous-tableau 1d avec un pointeur */
int main(void) {
    double tab[N]; // declaration du tableau 1D initial
    double *ptrd=NULL; // pointeur sur le même type
                        // que les éléments du tableau

    int i;

    for (i = 0 ; i < N ; i++) {
        tab[i] = (double) i ; // remplissage du tableau
    }
}

```

MNI

179

2011-2012

```

ptrd = tab + 3; // équivaut à ptrd=&tab[3]

// affichage du tableau et du sous tableau
for (i = 0 ; i < N ; i++) {
    printf(" tab[%d] = %g", i, tab[i]);
    if (i < N - 3 ) { // au delà, sortie du tableau initial
        printf(", ptrd[%d] = %g", i, ptrd[i]);
    }
    printf("\n") ;
}
// on peut meme revenir en arriere
printf("ptrd[-1]=%g\n",ptrd[-1]);

exit(0) ;
}

```

Résultat à l'exécution :

```

tab[0] = 0, ptrd[0] = 3
tab[1] = 1, ptrd[1] = 4
tab[2] = 2, ptrd[2] = 5
tab[3] = 3, ptrd[3] = 6
tab[4] = 4, ptrd[4] = 7
tab[5] = 5, ptrd[5] = 8
tab[6] = 6, ptrd[6] = 9
tab[7] = 7
tab[8] = 8
tab[9] = 9
ptrd[-1]=2

```

8.4.5 Tableaux 2D et pointeurs

Un tableau 2D correspond à un tableau de tableau :

```
int tab2d[2][3] = {{1,2,3},{4,5,6}} ;
```

Un tableau 2D correspond donc à un **pointeur de pointeur** :

– tab2d est un **pointeur sur un pointeur d'entier**.

– Sa valeur constante est l'adresse du premier élément du tableau :

tab2d correspond à &tab2d[0]

– tab2d[i] pour i=0,1 est aussi un pointeur constant.

– Sa valeur constante est l'adresse du premier élément de la ligne i du tableau :

tab2d[i] correspond à *(tab2d+i) et donc à &tab2d[i][0].

Donc : *(* (tab2d+i)+j) correspond à tab2d[i][j].

```

/* fichier tab2d_defer.c */
#include <stdio.h>
#include <stdlib.h>
#define N 3 // taille fixe dans la 1ere direction
#define P 4 // taille fixe dans la 2eme direction
int main(void) {
    int tab2[N][P]; // tableau 2D de taille 12

    for (int i=0; i < N; i++) {
        for (int j=0; j < P; j++) {
            tab2[i][j]=i*j;
            printf("%d ", tab2[i][j]);
        }
        printf("\n");
    }
}

```

```
// affichage de tab2[1][2] de 3 manieres:
printf("%d\n", tab2[1][2]);
printf("%d\n", (*(tab2+1))[2]);
printf("%d\n", *(* (tab2+1)+2));
exit(0);
}
```

Résultat à l'exécution :

```
0 0 0 0
0 1 2 3
0 2 4 6
2
2
2
```

8.5 Fonctions et tableaux

8.5.1 Passage de tableaux de taille fixe (pour le compilateur)

Passage de tableau : transmission d'un tableau d'une fonction à une autre.

Dans le cas d'un tableau de taille fixe, cette taille est une expression constante

- qui est soit codée en dur au sein du programme (**à éviter**),
- soit spécifiée par une directive `#define` du préprocesseur (**préférable**)

Un tableau étant un **pointeur** sur le premier élément du tableau :

⇒ **passage par copie d'adresse** (celle du premier élément)

⇒ les valeurs stockées dans le tableau peuvent être **modifiées** de la fonction appelante vers la fonction appelée et vice versa.

Inconvénient : changer la taille nécessite une recompilation

8.5.2 Exemple de passage d'un tableau 1D de taille fixe

```
/* fichier C99/tab1d-fixe.c */
#include <stdio.h>
#include <stdlib.h>
#define N 4 // taille fixe du tableau
void init_et_print(int t1d[N]); // prototype de la fonction
void init_et_print(int t1d[N]) { // definition de la fonction
    // tableau 1D ^^^^^ : pointeur
    for (int i=0; i < N; i++) {
        t1d[i] = i - 2; // affectation
        printf("%2d ", t1d[i]); // affichage
    }
    printf("\n");
    return; // fonction sans retour
}
```

```
int main(void) {
    int tab1d[N]; // taille fixe de N=4

    init_et_print(tab1d); // passage par copie d'adresse
    // tab1d ^^^^^ est equivalent a &tab1d[0]
    // le tableau a effectivement ete modifie dans le main:
    for (int i=0; i < N; i++) {
        printf("dans main : tab1d[%d]=%d\n", i, tab1d[i]);
    }
    exit(EXIT_SUCCESS);
}
```

Résultat à l'exécution :

```
-2 -1 0 1
dans main : tab1d[0]=-2
dans main : tab1d[1]=-1
dans main : tab1d[2]=0
dans main : tab1d[3]=1
```

8.5.3 Exemple de passage d'un tableau 2D de taille fixe

```

/* fichier C99/tab2d-fixe.c */
#include <stdio.h>
#include <stdlib.h>
#define N 4 // taille fixe dans la premiere direction
#define P 8 // taille fixe dans la deuxieme direction
void init_et_print(int t2d[N][P]); // prototype de la fonction
void init_et_print(int t2d[N][P]) { // definition de la fonction
    // tableau 2D ^^^^^^^^^ : pointeur de pointeur
    for (int i=0; i < N; i++) {
        for (int j=0; j < P; j++) {
            t2d[i][j]=i-j; // affectation
            printf("%2d ",t2d[i][j]); // affichage
        }
        printf("\n");
    }
    return; // fonction sans retour
}

```

MNI

188

2011-2012

8 Tableaux

Langage C

8.5 Fonctions et tableaux

```

int main(void) {
    int tab2d[N][P]; // taille fixe de NxP=32

    init_et_print(tab2d); // passage par copie d'adresse
    // tab2d ^^^^^ est equivalent a &tab2d[0]
    // le tableau a effectivement ete modifie dans le main:
    printf("dans main : tab2d[0][4]=%d\n",tab2d[0][4]);
    exit(EXIT_SUCCESS);
}

```

Résultat à l'exécution :

```

 0 -1 -2 -3 -4 -5 -6 -7
 1  0 -1 -2 -3 -4 -5 -6
 2  1  0 -1 -2 -3 -4 -5
 3  2  1  0 -1 -2 -3 -4
dans main : tab2d[0][4]=-4

```

MNI

189

2011-2012

8 Tableaux

Langage C

8.5 Fonctions et tableaux

8.5.4 Passage de tableau de taille inconnue à la compilation

Taille inconnue ⇒ **utiliser les tableaux automatiques**

⇒ la taille peut être choisie à l'exécution du programme

Le passage se fait toujours par copie d'adresse

⇒ modification possible du tableau dans la fonction appelante.

Attention à la syntaxe : dans la liste des arguments de la fonction, il faut déclarer la taille du tableau **AVANT** le tableau lui-même.

Exemple : cas d'un tableau 2D

la fonction doit avoir un entête du type :

```
void init_et_print(int n, int p, int t[n][p])
```

⇒ déclarer n et p avant t dans les arguments de la fonction

MNI

190

2011-2012

8.5.5 Exemple de passage d'un tableau 1D de taille variable

```

/* fichier C99/tab1d-var.c */
#include <stdio.h>
#include <stdlib.h>
//----- attention: norme C99 -----
// passage en argument d'un tableau 1d de taille variable
// => il faut aussi passer la taille du tableau à la fct
// => il ft déclarer la taille du tabl. AVANT le tableau
void init_et_print(const int n, int t[n]);
// ATTENTION: ^^^^^ n non modifiable dans la fonction
void init_et_print(const int n, int t[n]) {
    for (int i=0; i < n; i++) {
        t[i] = i - 2; // affectation
        printf("%2d ",t[i]); // affichage
    }
    printf("\n");
    return; // fonction sans retour
}

```

MNI

191

2011-2012


```
int main(void) {
    int n;
    printf("Entrer n: ");
    scanf("%d", &n);
    int tab[n]; // tableau de taille variable, definie par l'utilisateur
    init_et_print(n,tab); // passage par copie d'adresse de tab
    printf("dans main : tab[n-1]=%d\n",tab[n-1]);
    exit(EXIT_SUCCESS);
}
```

```
Entrer n: 4
-2 -1 0 1
dans main : tab[n-1]=1
```

MNI

192

2011-2012

8 Tableaux

Langage C

8.5 Fonctions et tableaux

}

```
int main(void) {
    int n, p;
    printf("Entrer n et p: ");
    scanf("%d %d", &n, &p);
    int tab[n][p]; // taille variable, definie par l'utilisateur
    init_et_print(n,p,tab); // passage par copie d'adresse de tab
    printf("dans main : tab[n-1][p-1]=%d\n",tab[n-1][p-1]);
    exit(EXIT_SUCCESS);
}
```

```
Entrer n et p: 2 8
0 -1 -2 -3 -4 -5 -6 -7
1 0 -1 -2 -3 -4 -5 -6
dans main : tab[n-1][p-1]=-6
```

MNI

194

2011-2012

8.5.6 Exemple de passage d'un tableau 2D de taille variable

```
/* fichier C99/tab2d-var.c */
#include <stdio.h>
#include <stdlib.h>
//----- attention: norme C99 -----
// passage en argument d'un tableau 2d de taille variable
// => il faut aussi passer la taille du tableau à la fct
// => il ft déclarer la taille du tabl. AVANT le tableau
void init_et_print(const int n, const int p, int t[n][p]);
// ATTENTION: ^^^^^ n et ^^^^^ p non modifiables dans la fonction
void init_et_print(const int n, const int p, int t[n][p]) {
    for (int i=0; i < n; i++) {
        for (int j=0; j < p; j++) {
            t[i][j]=i-j; // affectation
            printf("%2d ",t[i][j]); // affichage
        }
        printf("\n");
    }
    return; // fonction sans retour
}
```

MNI

193

2011-2012

8 Tableaux

Langage C

8.5 Fonctions et tableaux

8.5.7 Limite des tableaux automatiques

Les tableaux automatiques sont la solution à utiliser dans la plupart des cas

Comme toutes les variables considérées jusqu'ici, ces tableaux sont visibles dans le programme :

- après leur déclaration et au sein du **bloc dans lequel ils sont déclarés**,
- dans les **fonctions appelées** dans ce bloc, pourvu qu'ils soient passés comme argument.

Limitation :

les tableaux automatiques, s'ils sont déclarés dans une fonction, ne sont **pas visibles dans la fonction appelante**.

Remarque :

En fait, ces tableaux sont déclarés sur la « **pile** » : c'est-à-dire la partie de la mémoire gérée **automatiquement** par l'ordinateur.

MNI

195

2011-2012

```

/* fichier C99/tab-pile2.c */
#include <stdio.h>
#include <stdlib.h>
// ----- attention: norme C99 -----
// limite des tableaux automatiques:
// declares dans une fonction ils ne sont PAS
// visibles dans la fonction appelante
void print_1D(const int n, const int t[n]);
//      n ^^^^^ et t ^^^^^ non modifiables
int * retour_tab(const int nn);
// ^ retourne un pointeur de int (tableau 1D de int)

void print_1D(const int n, const int t[n]) {
    printf("impression du tableau dans print_1d:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", t[i]) ; // pas de modification de t
    }
    printf("\n");
    return; // fonction sans retour
}

```

MNI

196

2011-2012

8 Tableaux

Langage C

8.5 Fonctions et tableaux

```

}
exit(0) ;
}

```

Avertissement à la compilation :

```

tab-pile2.c: In function 'retour_tab':
tab-pile2.c:23: attention : cette fonction retourne
l'adresse d'une variable locale

```

Problème (aléatoire) à l'exécution :

```

entrer la taille n du tableau: 5
impression du tableau dans print_1d:
0 1 2 3 4
impression du tableau dans le main:
-1228844832 32767 438773941 62 4

```

MNI

198

2011-2012

```

int * retour_tab(const int nn){
    int tab[nn]; // declaration locale de tab
    for (int i=0; i<nn; i++) {
        tab[i] = i; // affectation de tab
    }
    print_1D(nn, tab);
    return tab; // fonction retournant un tableau de int
}

int main(void) {
    int n, *t=NULL; // declaration de t sans initialisation
    printf("entrer la taille n du tableau: \n");
    scanf("%d", &n);

    t = retour_tab(n); // appel de la fonction tab_var

    printf("impression du tableau dans le main:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", t[i]) ; // retourne n'importe quoi !
    }
}

```

MNI

197

2011-2012

8 Tableaux

Langage C

8.5 Fonctions et tableaux

Ce type de problème survient aussi dans le cas suivant :

- un fichier de données contient la taille d'un tableau et les valeurs de ses éléments,
- on veut écrire une fonction qui lit ce fichier, « alloue » le tableau et le remplit,
- cette fonction est appelée (par exemple) dans `main`.

Problème : impossible d'utiliser le tableau dans le `main` puisqu'il n'y est pas visible.

Solution : utiliser un tableau dynamique \Rightarrow allocation dynamique.

Remarque :

Les tableaux dynamiques sont déclarés sur le « **tas** » : c'est-à-dire la partie de la mémoire gérée **manuellement** par le programmeur. Le tas permet d'assurer une **visibilité** des variables.

MNI

199

2011-2012

9 Allocation dynamique (sur le tas)

9.1 Motivation

Du point de vue de l'allocation de mémoire, on distingue trois types de tableaux :

- **tableaux statiques** : occupent un emplacement bien défini avant exécution
- **tableaux automatiques** : pas d'emplacement défini avant exécution
⇒ alloués/désalloués "automatiquement" (sur la **pile** (*stack* en anglais))
- **tableaux dynamiques** : pas d'emplacement défini avant exécution
⇒ alloués/désalloués "manuellement" (sur le **tas** (*heap* en anglais))

Avantage des tableaux dynamiques :

- leur taille peut varier pendant l'exécution du programme comme pour les tableaux automatiques,
- les tableaux dynamiques peuvent être alloués dans une fonction et retournés dans une autre contrairement aux tableaux automatiques.

C'est au programmeur de se charger de l'allocation et de la libération de la mémoire dynamique

MNI

200

2011-2012

Prototype de `calloc` :

```
void *calloc(size_t nb_bloc, size_t taille) ;
```

- Deux arguments **taille** : nombre d'octets à allouer.
 - **nb_bloc** : nombre de blocs consécutifs de **taille** octets à allouer,
 - **taille** : nombre d'octets par bloc.
- Une valeur de retour du type **void *** (**pointeur** sur `void`) :
 - l'adresse de l'emplacement alloué si tout se passe bien,
 - le pointeur **NULL** en cas de problème.

La fonction `calloc` initialise tous les octets alloués à zéro binaire (OK pour les entiers, problème possible pour les réels)

Exemple : allocation d'un tableau 1D de 10 doubles

```
double *ptr = NULL ;
ptr = (double *) calloc(10, sizeof(double)) ;
```

MNI

202

2011-2012

9.2 Allocation dynamique avec `malloc` ou `calloc`

Deux fonctions standard permettent d'allouer un espace mémoire sur le tas.

Leur prototype est dans le fichier : `stdlib.h`

⇒ directive préprocesseur (compilation) : `#include <stdlib.h>`

Prototype de `malloc` : `void *malloc(size_t taille) ;`

- Un argument **taille** : nombre d'octets à allouer.
- Une valeur de retour du type **void *** (**pointeur** sur `void`) :
 - l'adresse de l'emplacement alloué si tout se passe bien,
 - le pointeur **NULL** en cas de problème.

Exemple : allocation d'un tableau 1D de 10 doubles

```
double *ptr = NULL ;
ptr = (double *) malloc(10*sizeof(double)) ;
```

Le pointeur peut être ensuite utilisé avec le formalisme tableau (`ptr[0]...ptr[9]`)

Noter : conversion de **void *** en **double *** et utilisation de `sizeof`

MNI

201

2011-2012

9.3 Libération de la mémoire allouée avec `free`

La gestion de la mémoire dynamique (le « tas ») est **manuelle**

⇒ la libération de l'espace alloué est à faire par le programmeur.

La fonction standard `free` permet de libérer la mémoire allouée dynamiquement.

(son prototype est dans le fichier : `stdlib.h`)

Prototype de `free` : `void free(void *adr) ;`

- Un argument **adr** : adresse de l'emplacement à libérer, (**adr** est un pointeur qui aura été initialisé par `malloc` ou `calloc`)
- Aucune valeur de retour.

Exemple : allocation puis libération d'un tableau 1D de 10 doubles

```
double *ptr = NULL ;
ptr = (double *) calloc(10, sizeof(double)) ;
...
free(ptr) ;
ptr = NULL ; (pour plus de sécurité)
```

MNI

203

2011-2012

9.4 Attention aux fuites de mémoire

```
int *adr = NULL ;
n = 1 ;
adr = (int *) calloc(100, sizeof(int)) ;
adr = &n ; /* fuite de memoire */
```

Plus aucun pointeur ne pointe vers l'espace alloué par `calloc`!

⇒ cette partie de la mémoire est perdue pour le restant du programme,
il est impossible d'y accéder à nouveau.

(il aurait fallu libérer la mémoire allouée avant de réaffecter `adr`)

⇒ particulièrement grave dans les boucles.

Des fuites de mémoires massives peuvent provoquer un plantage du programme.

Remarque : certains langages (Java, mais pas C) ont des mécanismes de *garbage collector* pour détecter ces espaces perdus.

```
int main(void) {
    int n, *pti = NULL; // pointeur sur entier local => NULL
    printf("Entrer n ");
    scanf("%d", &n);

    pti = alloc_et_init(n); // affectation par retour fonction
    if (pti != NULL) { // si l'allocation marche...
        for (int i=0; i < n; i++) {
            printf("%d ", pti[i]); // ... affichage du tableau 1D
        }
    }
    printf("\n");
    free(pti); // libération de l'espace
    pti=NULL; // sécurité
    exit(0);
}
```

Entrer n 6

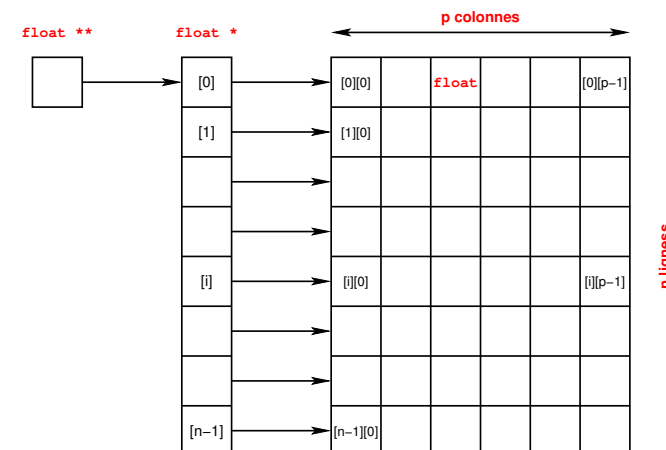
0 1 2 3 4 5

9.5 Exemple d'allocation dynamique d'un tableau 1D

```
/* fichier alloc-tab1d2.c */
#include <stdio.h>
#include <stdlib.h>
/* ----- allocation dynamique d'un tableau 1D ----- */
int * alloc_et_init(const int n) {
    int *p=NULL; // pointeur sur entier local => NULL
    p=(int *)calloc(n, sizeof(int)); // affectation pointeur
    if (p == NULL) { // si l'affectation ne marche pas...
        printf("Erreur d'allocation\n");
        return p; // ... retour dans main avec le pointeur nul
    }
    for (int i=0; i < n; i++) {
        p[i]=i; // pointeur sur entier = tableau 1D de int
    }
    return p; // fonction retourne le pointeur (tableau 1D)
}
```

9.6 Exemple d'allocation dynamique d'un tableau 2D

Tableau 2D en C ⇒ pointeur de pointeur



```

/* fichier alloc-tab2d.c */
#include <stdio.h>
#include <stdlib.h>
int ** alloc2d(const int n, const int p) {
    int *pespace=NULL, **ptab=NULL;

// 1) Allocation de l'espace pour la matrice
    pespace=(int *)calloc(n*p, sizeof(int));
    if (pespace == NULL) { // pb allocation
        return ptab;
    }

// 2) Allocation du vecteur de pointeurs
    ptab=(int **)calloc(n, sizeof(int *));
    if (ptab == NULL) { // pb allocation
        return ptab;
    }

```

MNI

208

2011-2012

```

        for (int j=0; j < p; j++) {
            pti[i][j]=i*j;
            printf("%d ",pti[i][j]);
        }
        printf("\n");
    }
}
free(pti[0]); // libération de la matrice
free(pti); // libération du vecteur
pti=NULL; // sécurité
exit(0);
}

```

Entrer n et p 3 6

0 0 0 0 0 0

0 1 2 3 4 5

0 2 4 6 8 10

MNI

210

2011-2012

```

// 3) Affectation au debut de chaque ligne
    for (int i=0; i < n; i++) {
        ptab[i]=&pespace[i*p];
        // ou pespace + i*p
    }
    return ptab; // !! ptab est déclaré sur le tas
}

int main(void) {
    int n, p, **pti = NULL; /* initialisation à NULL */

    printf("Entrer n et p ");
    scanf("%d %d",&n, &p);
    pti=alloc2d(n, p);
    if (pti != NULL) { // allocation OK
        for (int i=0; i < n; i++) {

```

MNI

209

2011-2012

9.7 La bibliothèque libmnitab

Dans le cas des tableaux sur le tas : utiliser la bibliothèque **libmnitab**

⇒ directive préprocesseur `#include "mnitab.h"`

⇒ compilation avec `gcc+mni` ou `gcc+mni-c99`

⇒ édition des liens avec `-lmnitab`

Cette bibliothèque contient de nombreuses fonctions permettant de gérer la mémoire dynamique.

Exemples :

– Allocation et libération de tableau 1D de doubles :

`double *double1d(int n)` pour allouer l'espace et

`void double1d_libere(double *vec)` pour libérer l'espace

– Allocation et libération de tableaux 2D de floats :

`float **float2d(int n, int p)` pour allouer l'espace et

`void float2d_libere(float **mat)` pour libérer l'espace

– d'autres fonctions de calcul de min ou de max...

MNI

211

2011-2012

9.7.1 Exemple de passage d'un tableau dynamique 1D

```

/* fichier C99/tab1d-dyn.c */
#include <stdio.h>
#include <stdlib.h>
#include "mnitab.h" // utilisation de la bibliotheque libmnitab
// passage en argument d'un tableau 1d dynamique
// => il faut aussi passer la taille du tableau à la fct
void init_et_print(int n, double *t);
void init_et_print(int n, double *t) {
    for (int i=0; i < n; i++) {
        t[i] = (double) 1/(i + 1.); // affectation
        printf("%4g ",t[i]); // affichage
    }
    printf("\n");
    return; // fonction sans retour
}

```

MNI

212

2011-2012

9 Allocation dynamique (sur le tas)

Langage C

9.7 La bibliothèque libmnitab

9.7.2 Exemple de passage d'un tableau dynamique 2D

```

/* fichier C99/tab2d-dyn.c */
#include <stdio.h>
#include <stdlib.h>
#include "mnitab.h" // utilisation de la bibliotheque libmnitab
// passage en argument d'un tableau 2d dynamique
// => il faut aussi passer la taille du tableau à la fct
void init_et_print(int n, int p, double **t);
void init_et_print(int n, int p, double **t) {
    for (int i=0; i < n; i++) {
        for (int j=0; j < p; j++) {
            t[i][j]=(double) 1./(i+j+1.); // affectation
            printf("%10g ",t[i][j]); // affichage
        }
        printf("\n");
    }
    return; // fonction sans retour
}

```

MNI

214

2011-2012

```

int main(void) {
    int n;
    double *tab=NULL; // declaration d'un pointeur sur un double
    printf("Entrer n: ");
    scanf("%d", &n);
    tab = double1d(n); // allocation dynamique au moyen de la bibliotheque
    init_et_print(n,tab); // passage par copie d'adresse de tab
    printf("dans main : tab[n-1]=%g\n",tab[n-1]);
    double1d_libere(tab); // liberation de l'espace memoire
    tab = NULL; // par precaution
    exit(EXIT_SUCCESS);
}

```

```

Entrer n: 4
1 0.5 0.333333 0.25
dans main : tab[n-1]=0.25

```

MNI

213

2011-2012

9 Allocation dynamique (sur le tas)

Langage C

9.7 La bibliothèque libmnitab

```

int main(void) {
    int n, p;
    double **tab=NULL; // declaration d'un double pointeur sur un double
    printf("Entrer n et p: ");
    scanf("%d %d", &n, &p);
    tab = double2d(n,p); // allocation dynamique d'un tableau 2d de double
    init_et_print(n,p,tab); // passage par copie d'adresse de tab
    printf("dans main : tab[n-1][p-1]=%g\n",tab[n-1][p-1]);
    double2d_libere(tab); // liberation de la memoire allouee
    tab = NULL; // dissociation du pointeur
    exit(EXIT_SUCCESS);
}

```

```

Entrer n et p: 3 4
1 0.5 0.333333 0.25
0.5 0.333333 0.25 0.2
0.333333 0.25 0.2 0.166667
dans main : tab[n-1][p-1]=0.166667

```

MNI

215

2011-2012

9.8 Bilan sur l'allocation de mémoire

Quand utiliser l'allocation automatique (sur la pile) ?

Si le tableau est utilisé :

- dans la fonction où il est déclaré,
- ou dans une fonction appelée par celle-ci,

⇒ préférer les tableaux de taille variables sur la pile (`int tab[n][p]`)

Quand utiliser l'allocation dynamique (sur le tas) ?

On peut TOUJOURS utiliser des tableaux dynamiques.

C'est indispensable si le tableau est utilisé dans la fonction appelant la fonction où le tableau est défini

⇒ utiliser les allocations dynamiques (`int **tab`)

Attention : les deux types de déclarations **ne sont pas équivalentes**

⇒ ne pas mélanger les syntaxes.

Pour les tableaux dynamiques : penser à utiliser la bibliothèque **libmmitab**.

MNI 216 2011-2012
10 Chaînes de caractères Langage C 10.2 Exemple de tableaux de caractères de taille fixe

10 Chaînes de caractères

10.1 Définition et usage

En C, il n'y a pas de type « chaîne de caractères » en C

⇒ on utilise des **tableaux de caractères**

qui se terminent par le caractère fin de chaîne : `\0`.

Ces tableaux peuvent être **statiques**, **automatiques** ou **dynamiques**.

Il existe une **bibliothèque standard** contenant des fonctions permettant de manipuler les tableaux de caractères.

Dans la pratique, les chaînes de caractères les plus répandues sont :

- le premier argument des fonctions `printf` et `scanf`,
- les noms des fichiers manipulés par le programme,
- etc ...

MNI 217 2011-2012
10 Chaînes de caractères Langage C 10.2 Exemple de tableaux de caractères de taille fixe

10.2 Exemple de tableaux de caractères de taille fixe

```
/* fichier char-elem_stat.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h> // contient le prototype de strlen, etc...
#define N 5
// ----- tableaux de caracteres de taille fixe -----
int main(void) {
    // declaration sans affectation:
    char cfixe1[N]; // tableau de N-1 caracteres + \0
    // Declaration avec affectation:
    char cfixe2[N]={'O','u','i','\0'}; // doit inclure \0
    // Declaration avec affectation (methode conseillée):
    char cfixe3[N]= "Non"; // sans préciser le \0
```

MNI 218 2011-2012

```
// Affectation du premier tableau a l'aide d'une boucle:
for(int i=0;i<N-1;i++){ // attention: indice de 0 a N-2
    cfixe1[i] = 'z';
    printf("%c", cfixe1[i]); // format %c pour un seul caractere
}
// Inclusion du caractere de fin de chaine (\0) - impératif:
cfixe1[N-1] = '\0';
// Affichage de la taille (sans \0) a l'aide de la fonction strlen:
printf("\nTaille du 1er tableau: %d\n", (int) strlen(cfixe1));
printf("Taille du 2eme tableau: %d\n", (int) strlen(cfixe2));
printf("Taille du 3eme tableau: %d\n", (int) strlen(cfixe3));
// Affichage des tableaux de caracteres (format %s pour une chaine):
printf("Affichage du 1er tableau: %s\n", cfixe1);
printf("Affichage du 2eme tableau: %s\n", cfixe2);
printf("Affichage du 3eme tableau: %s\n", cfixe3);
exit(EXIT_SUCCESS);
}
```

MNI 219 2011-2012

10.3 Exemple de tableaux de caractères de taille quelconque

Résultat à l'exécution :

```

zzzz
Taille du 1er tableau:  4
Taille du 2eme tableau:  3
Taille du 3eme tableau:  3
Affichage du 1er tableau:  zzzz
Affichage du 2eme tableau:  Oui
Affichage du 3eme tableau:  Non

```

MNI

220

2011-2012

10 Chaînes de caractères

Langage C

10.3 Exemple de tableaux de caractères de taille quelconque

```

    cvar3[i] = 'd';
    cvar4[i] = 'a';
}
// Inclusion du caractere de fin de chaine (\0) - impératif
cvar3[nb] = '\0';
cvar4[nb] = '\0';
// Affichage de la taille (sans \0) a l'aide de la fonction strlen:
printf("Taille du premier tableau:  %d\n", (int) strlen(cvar1));
printf("Taille du deuxieme tableau:  %d\n", (int) strlen(cvar2));
printf("Taille du tableau dynamique:  %d\n", (int) strlen(cvar3));
printf("Taille du tableau automatique:  %d\n", (int) strlen(cvar4));
// Affichage des tableaux de caracteres:
printf("Affichage du premier tableau:  %s\n", cvar1);
printf("Affichage du deuxieme tableau:  %s\n", cvar2);
printf("Affichage du tableau dynamique:  %s\n", cvar3);
printf("Affichage du tableau automatique:  %s\n", cvar4);
free(cvar3); // liberation de la memoire allouee dynamiquement

```

MNI

222

2011-2012

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h> // contient prototype de strlen, etc...
// ----- tableaux de caracteres de taille quelconque -----
int main(void){
    int nb; // nombre de caracteres
    char *cvar1 = "Bonjour!"; // taille calculée a l'initialisation
    char cvar2[] = "Re-bonjour"; // taille calculée a l'initialisation
    char *cvar3 = NULL; // tableau dynamique de caracteres
    printf("Nombre de caracteres (tableaux dynamique et automatique):\n");
    scanf("%d", &nb); // choix du nombre de caracteres par l'utilisateur
    cvar3 = (char *) calloc(nb+1, sizeof(char)); // allocation dynamique
    char cvar4[nb+1]; // declaration tardive d'un tableau automatique
    // Affectation des tableaux a l'aide d'une boucle:
    for(int i=0; i<nb; i++){ // attention: indice de 0 a nb

        cvar3=NULL; // par precaution
        exit(EXIT_SUCCESS);
    }
}

```

Résultat à l'exécution :

```

Nombre de caracteres (tableaux dynamique et automatique):
5
Taille du premier tableau:  8
Taille du deuxieme tableau:  10
Taille du tableau dynamique:  5
Taille du tableau automatique:  5
Affichage du premier tableau:  Bonjour!
Affichage du deuxieme tableau:  Re-bonjour
Affichage du tableau dynamique:  ddddd
Affichage du tableau automatique:  aaaaa

```

MNI

221

2011-2012

10 Chaînes de caractères

Langage C

10.3 Exemple de tableaux de caractères de taille quelconque

MNI

223

2011-2012

10.4 Déclaration et affectation des chaînes de caractères

10.4.1 Chaîne de longueur fixe

- `char st1[4] = "oui" ;` (méthode conseillée)
pas besoin de spécifier le caractère de fin de chaîne `\0`
- `char st1[4] = {'o', 'u', 'i', '\0'} ;`
il est impératif d'inclure le caractère de fin de chaîne `\0`

10.4.2 Chaîne de longueur calculée à l'initialisation

- `char *st2 = "oui" ;` (méthode conseillée)
- `char st2[] = "oui" ;` (méthode conseillée)

NB : ceci n'est possible qu'à la déclaration, car les chaînes étant des tableaux, on ne peut pas faire d'affectation globale (c'est-à-dire de tous les éléments en une seule instruction).

L'utilisation de tableaux automatiques et dynamiques est possible.

L'utilisation de boucles pour l'affectation peut s'avérer peu pratique.

MNI

224

2011-2012

10.5 Manipulation des chaînes de caractères

Prototypes des fonctions dans `string.h`

⇒ directive préprocesseur `#include <string.h>`

10.5.1 Longueur d'une chaîne avec `strlen`

Prototype : `size_t strlen(const char *s) ;`

Calcul de la longueur d'une chaîne sans le caractère fin de chaîne.

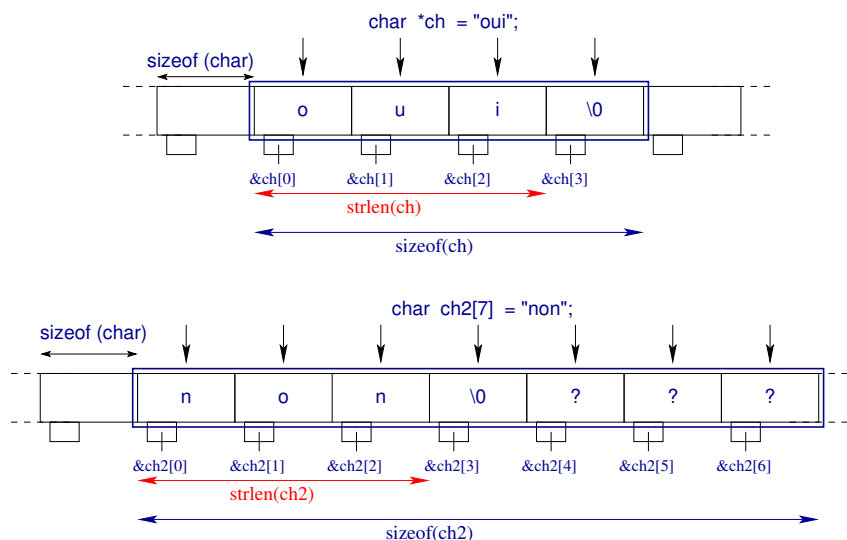
Exemple :

```
char *ch="oui";
char ch2[7]="non";
printf("%d\n", sizeof(ch)); // => 4 ('o' 'u' 'i' '\0')
printf("%d\n", strlen(ch)); // => 3 ('o' 'u' 'i')
printf("%d\n", sizeof(ch2)); // => 7 ('n' 'o' 'n' + '\0'+3)
printf("%d\n", strlen(ch2)); // => 3 ('n' 'o' 'n')
```

MNI

225

2011-2012



MNI

226

2011-2012

10.5.2 Concaténation de chaînes avec `strcat`

Prototype :

```
char *strcat(char *dest, const char *source) ;
```

Concatène (ajoute) la chaîne `source` à la chaîne `dest` et renvoie un pointeur sur `dest`. Gère le caractère `\0`.

Attention : `dest` doit être de longueur suffisante au risque de segmentation fault. La contrainte sur la taille de `dest` est : `sizeof(dest) >= strlen(dest) + strlen(source) + 1`

Exemple :

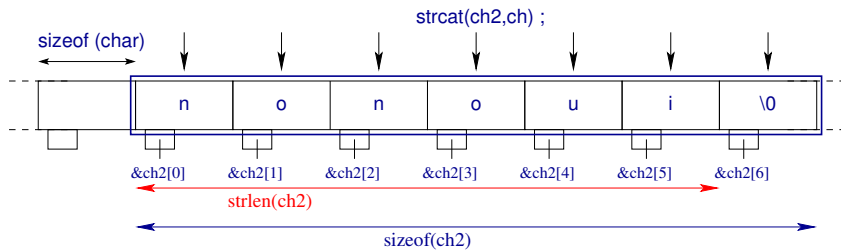
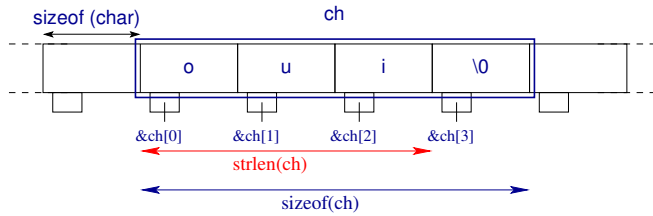
```
strcat(ch2, ch); // concatenation
printf("%s\n", ch2); // => nonoui
printf("%d\n", strlen(ch2)); // => 6
```

Noter le format `%s` dans `printf`

MNI

227

2011-2012



10.5.3 Copie d'une chaîne avec strcpy

Prototype :

```
char *strcpy(char *dest, const char *source) ;
```

Copie la chaîne **source** dans la chaîne **dest** et renvoie un pointeur sur **dest**. La valeur de retour est d'intérêt limité.

Attention : **dest** doit être de longueur suffisante au risque de segmentation fault. La contrainte sur la taille de **dest** est : $\text{sizeof}(\text{dest}) \geq \text{strlen}(\text{dest}) + \text{strlen}(\text{source}) + 1$

Attention : la fonction `strcpy` ne gère pas le caractère `\0`. Le programmeur doit s'assurer qu'il est présent dans **source** avant d'effectuer la copie.

Exemple :

```
char *ch="oui"; // chaine de 3 caracteres + \0
char ch2[7]; // chaine de taille totale 7
strcpy(ch2, ch); // copie
printf("%s\n", ch2); // => oui
```

10.5.4 Comparaison de chaînes avec strcmp

Prototype :

```
int strcmp(const char *s1, const char *s2) ;
```

Renvoie un **entier positif**, **nul** ou **négatif** si **s1** est plus grand, égal, plus petit que **s2** (dans l'ordre lexicographique).

Exemple :

```
char *ch="oui";
char ch2[7]="non";
printf("%d\n", strcmp(ch, ch2)); // => 1 ("oui" > "non")
```

10.5.5 Recherche d'un caractère dans une chaîne avec strchr

Prototype : `char *strchr(const char *s, int c) ;`

Renvoie un **pointeur de caractère** vers la première occurrence de **c** dans la chaîne **s**, ou **NULL** si **c** n'est pas trouvé.

Exemple :

```
char *ch="oui";
char *pc;
pc = strchr(ch, 'u');
printf("%d\n", pc - ch); // => 1 (car ch[1]='u')
```

10.5.6 Recherche d'une sous-chaîne dans une chaîne avec `strstr`

Prototype :

```
char *strstr(const char *foin, const char *aiguille) ;
```

Renvoie un **pointeur** vers la première occurrence de la sous-chaîne **aiguille** dans la chaîne **foin**, ou **NULL** si **aiguille** n'est pas trouvée.

Exemple :

```
char *ch2="nonoui";
char *aig="nou", *pc;
pc = strstr(ch2, aig);
printf("%d\n", pc - ch2); // => 2
```

MNI 232 2011-2012
11 Entrées-sorties Langage C 11.1 Introduction

11.1.1 Rappel : les fonctions `printf` et `scanf`

Les fonctions **printf** et **scanf** sont des fonctions d'entrée-sortie standard.

Leur prototype est dans le fichier : `stdio.h`

⇒ directive préprocesseur (compilation) : `#include <stdio.h>`

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

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

- valeur de retour du type **int**
 - **printf** : le nombre de caractères écrits si tout se passe bien ou un nombre négatif en cas d'erreur.
 - **scanf** : le nombre de variables effectivement lues si tout se passe bien ou un nombre négatif en cas d'erreur.
- nombre variable d'arguments : `...`, dont au moins une chaîne de caractères

Remarque : si on ne s'intéresse pas aux valeurs de retour de ces fonctions, elles peuvent être appelées comme des fonctions sans retour.

MNI 234 2011-2012

11 Entrées-sorties

11.1 Introduction

Jusqu'à présent, nos programmes fonctionnaient en interactif, et utilisaient les entrées-sorties dites **standard** :

- lecture des entrées sur le **clavier** (`stdin`)
 - écriture des sorties sur l'**écran** (`stdout`)
- ⇒ utilisation de `printf` et `scanf` (prototypes dans `stdio.h`)

Pour utiliser des informations stockées de façon permanente, ou pour stocker des informations, on a besoin de manipuler des **fichiers**. En particulier :

- **ouvrir/fermer un fichier à partir d'un programme**,
 - **accéder au contenu du fichier** (de manière séquentielle ou directe),
 - **effectuer des opérations sur le fichier** (lecture/écriture).
- ⇒ utilisation de `fopen`, `fclose`, `fprintf`, `fscanf`, ... (prototypes dans `stdio.h`)

MNI 233 2011-2012
11 Entrées-sorties Langage C 11.1 Introduction

11.1.2 Exemple introductif

```
/* programme lecture-elem.c */
#include <stdio.h>
#include <stdlib.h>
//----- exemple entree/sortie -----
// on suppose qu'on dispose de fichiers contenant deux champs:
// premier champ: nom des etudiants
// deuxieme champ: notes des etudiants
int main(void) {
    int n;
    char fichier[80]; // nom externe du fichier
    FILE *pfich=NULL; // declaration du flux (nom interne du fichier)
    char nom[20]; // nom etudiant
    float note; // note etudiant
    printf("Quel fichier ouvrir ?\n"); // affichage a l'ecran
    scanf("%s", fichier); // lecture du nom de fichier (clavier)
```

MNI 235 2011-2012

```

pfich=fopen(fichier,"r"); // ouverture du fichier en lecture
if (pfich == NULL) { // en cas de probleme...
    exit(EXIT_FAILURE); // ... arreter l'execution
}
while(1) { // boucle (a priori) infinie
    // on utilise la valeur de retour de fscanf:
    n=fscanf(pfich,"%s %f", nom, &note); // lecture du fichier
    if (n == EOF) { // EOF = End Of File
        break; // on quitte la boucle en fin de fichier
    }
    printf("%s %g\n", nom, note); // affichage a l'ecran
}
fclose(pfich); // fermeture du flux
exit(EXIT_SUCCESS) ;
}

```

11.2 Type de fichiers et accès

On distingue deux types de fichiers :

- **fichier binaire** : fichier dans lequel les données sont écrites comme en mémoire.
- **fichier formaté** : c'est un fichier texte.

fichiers	formatés	binaires
	saisie et affichage	comme en mémoire
capacité	—	+
rapidité des E/S	—	+
précision conservée	—	+
portabilité	+	—

On distingue deux manières d'accéder aux informations du fichier :

- **accès séquentiel (le plus courant)** : on lit/écrit dans l'ordre dans lequel les informations apparaissent/apparaîtront.
- **accès direct** : on lit/écrit à un endroit particulier du fichier dont les enregistrements sont indexés.

accès aux données	séquentiel	direct
	lire/écrire dans l'ordre	enregistrements indexés

Par défaut l'accès est séquentiel.

L'accès direct se fait au moyen de fonctions `fseek`, `ftell`,...

11.3 Ouverture et fermeture d'un fichier

On associe un **nom externe de fichier** (une chaîne de caractères) à un **nom interne dans le programme** (un pointeur sur `FILE`). Le nom interne est aussi appelé **flux**.

11.3.1 Déclaration d'un flux

Un flux est un pointeur sur une structure de type `FILE`. `FILE` est le type d'un objet contenant toutes les informations nécessaire à la gestion d'un fichier

La déclaration d'un flux `pfich` se fait au moyen de l'instruction :

```
FILE *pfich = NULL; // pfich: nom interne ou flux
```

Important : initialiser `pfich` à `NULL`.

Remarque : aucun lien effectif à ce niveau entre le flux et le fichier.

11.3.2 Ouverture d'un flux avec fopen

Le lien entre nom interne et nom externe est fait à l'ouverture d'un flux.

Prototype :

```
FILE *fopen(const char *nom_externe, const char *mode) ;
```

Valeur de retour :

pointeur sur FILE (flux) si tout se passe bien et le pointeur NULL sinon.

⇒ toujours tester la valeur retournée pour s'assurer de la bonne ouverture du flux

Arguments :

- **nom_externe** : chaîne de caractères indiquant le nom externe du fichier,
- **mode** : chaîne de caractères indiquant le mode d'ouverture du fichier

valeur		position	ajouter (facultatif)
"r"	read	début du fichier	+ si mise à jour (lecture+écriture)
"w"	write	début du fichier	b si binaire
"a"	append	fin du fichier	

Mode "r" : erreur si le fichier n'existe pas

Mode "w" : le fichier est créé s'il n'existe pas, écrasé s'il existe

Mode "a" : le fichier est créé s'il n'existe pas ; écriture à la fin du fichier s'il existe

11.4 Entrée-sorties formatées

Les opérations sur les fichiers se font sous forme texte.

Il est recommandé d'utiliser un flux ouvert en mode texte (le cas par défaut).

11.4.1 Ecriture avec fprintf

```
int fprintf(FILE* stream, const char* format, ... ) ;
```

Appartient à la famille de printf : indiquer le flux en plus.

Valeur de retour : la fonction fprintf renvoie

- le nombre de caractères écrits si tout se passe bien,
- ou un nombre négatif en cas d'erreur.

Remarques :

- printf = fprintf(stdout, ...)
- Pour écrire des messages d'erreurs, il vaut mieux utiliser :
fprintf(stderr, "Erreur...") que printf("Erreur...")

11.3.3 Fermeture d'un flux avec fclose

Prototype : int fclose(FILE *stream) ;

11.3.4 Exemple d'ouverture/fermeture d'un fichier

```
/* programme fich.c */
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    FILE *pfich = NULL; // important: initialiser pfich a NULL
    pfich=fopen("../rep1/rep2/toto.txt", "w");
    if (pfich == NULL) { // en cas de probleme...
        printf("Le fichier ../rep1/rep2/toto.txt n'existe pas\n");
        exit(EXIT_FAILURE); // ... arreter l'execution
    }
    // ... instructions...
    fclose(pfich); // fermer le flux avant d'arreter l'execution
    exit(EXIT_SUCCESS);
}
```

11.4.2 Lecture avec fscanf

```
int fscanf(FILE* stream, const char* format, ... ) ;
```

Appartient à la famille de scanf : indiquer le flux en plus.

Valeur de retour : la fonction fscanf renvoie

- le nombre de variables attendues si tout se passe bien,
- un nombre négatif en cas d'erreur.
L'entier retourné peut être égal à la constante EOF (end of file) si l'on a atteint la fin du fichier.

Remarque : scanf = fscanf(stdin, ...)

11.4.3 Bilan sur les entrées-sorties formatées

Il existe deux familles de 3 fonctions en C pour lire/écrire :

- sur les entrée/sortie standards
- dans les fichiers
- dans les chaînes de caractères

stdout	<code>printf(format [,liste de variables])</code>
fichier	<code>fprintf(FILE* stream, format [,liste d'expressions])</code>
chaîne	<code>sprintf(char* string, format [,liste d'expressions])</code>

stdin	<code>scanf(format, liste d'adresses)</code>
fichier	<code>fscanf(FILE* stream, format, liste de pointeurs)</code>
chaîne	<code>sscanf(char* string, format, liste de pointeurs)</code>

11.5 Entrées-sorties non formatées (binaires)

Les opérations sur les fichiers se font sous forme binaire.

Il est fortement recommandé d'utiliser un flux ouvert en mode binaire.

11.5.1 Lecture avec fread

Prototype : `size_t fread(void *ptr, size_t taille, size_t nbloc, FILE *stream) ;`

La fonction **fread** :

- lit **nbloc** (le plus souvent 1) de taille **taille** dans le flux **stream** et les écrit à partir de l'adresse **ptr**.
- retourne le nombre de blocs effectivement lu
⇒ erreur si ce nombre est différent de **nbloc**.
Cette erreur peut être la rencontre de la fin du fichier.

11.5.2 Écriture avec fwrite

Prototype :

```
size_t fwrite(const void *ptr, size_t taille,
              size_t nbloc, FILE *stream) ;
```

La fonction **fwrite** :

- écrit **nbloc** (le plus souvent 1) de taille **taille** situés à l'adresse **ptr** dans le flux **stream**.
- retourne le nombre de blocs effectivement écrits
⇒ erreur si ce nombre est différent de **nbloc**.

11.6 Retour sur les formats d'entrée-sortie

(gabarit : w =largeur, p= précision)

entiers	
décimal	<code>%w[.p]d</code>
octal	<code>%wo</code>
hexadécimal	<code>%wx</code>
réels	
virgule fixe	<code>%w[.p]f</code>
virgule flottante (notation scientifique)	<code>%w[.p]e</code>
général (combinaison)	<code>%w[.p]g</code>
caractères	
caractères	<code>%w[.p]c</code>
chaîne	<code>%w[.p]s</code>

Voir la section entrées-sorties standard élémentaires.

11.7 Exemple de lecture de fichier formaté en C

La largeur **w** est facultative pour les formats %d, %f, %e, %g, %s
 Si la largeur **w** est insuffisante, elle est élargie pour écrire l'expression.
 Si le nombre de descripteurs \neq nb d'éléments de la liste d'E/S, le nb de
 descripteurs prime
 ⇒ risque d'accès mémoire non réservée

```
/* programme lecture.c */
#include <stdio.h>
#include <stdlib.h>
/* lecture du fichier donnees */
int main(void) {
char nom[20] ; /* limitation des chaînes à 20 caractères */
char article[50] ;
int nombre ;
float prix, dette ;
int n , ligne=1;
FILE *pf = NULL ; /* pointeur sur le flux d'entrée */
//char *fichier ="donnees"; /* nom du fichier formaté */
char fichier[80];
```

MNI 248 2011-2012
 11 Entrées-sorties Langage C 11.7 Exemple de lecture de fichier formaté en C

```
printf("Quel fichier ouvrir ?\n");
scanf("%s", fichier); // lecture du nom de fichier
pf=fopen(fichier,"r"); // ouverture du fichier
if (pf == NULL) { // pb d'ouverture
fprintf(stderr, "erreur ouverture du fichier %s\n", fichier) ;
exit (EXIT_FAILURE) ;
}
printf("Le fichier %s s'ouvre correctement\n", fichier) ;

while(1) { // boucle de lecture des données
n=fscanf(pf, "%s %s %d %f", nom, article, &nombre, &prix);
if (n == EOF) { // on quitte la boucle en fin de fichier
break;
}
}
```

MNI 250 2011-2012

MNI 249 2011-2012
 11 Entrées-sorties Langage C 11.7 Exemple de lecture de fichier formaté en C

```
if (n == 4) { /* si fscanf a réussi à convertir 4 variables */
dette = nombre * prix ;
printf("%s %s \t %2d x %6.2f = %8.2f\n", nom, article,
nombre, prix, dette);
ligne++;
} else {
fprintf(stderr, "problème fscanf ligne %d\n", ligne);
exit (EXIT_FAILURE) ;
}
}
/* sortie normale par EOF */
printf("fin de fichier %d lignes lues \n", ligne-1);
fclose(pf) ; /* fermeture du fichier */
exit(EXIT_SUCCESS) ;
}
```

MNI 251 2011-2012

```

donnees
dupond cafe 5 10.5
durand livre 13 60.2
jean disque 5 100.5
paul cafe 6 12.5
jean disque 4 110.75
julie livre 9 110.5

resultat
Quel fichier ouvrir ?
donnees
Le fichier donnees s'ouvre correctement
dupond cafe      5 x 10.50 = 52.50
durand livre     13 x 60.20 = 782.60
jean disque      5 x 100.50 = 502.50
paul cafe        6 x 12.50 = 75.00
jean disque      4 x 110.75 = 443.00
julie livre      9 x 110.50 = 994.50
fin de fichier 6 lignes lues

```

11.8 Fonctions supplémentaires

Lecture

```
char *fgets(char *s,
            int size, FILE *stream)
```

permet de lire `size` caractères dans la chaîne `s` à partir du flux `stream` (éventuellement `stdin`).

Contrairement à `fscanf(stream, "%s", ...)`, `fgets` ne s'arrête pas à la rencontre d'un caractère espace mais en fin de ligne.

La valeur de retour est l'adresse de la chaîne lue **quand tout s'est bien passé**, ou égale à `NULL` **en cas d'erreur**.

12 Structures ou types dérivés

12.1 Intérêt des structures

Tableau = agrégat d'objets de **même type** repérés par un **indice entier**

Structure = agrégat d'objets de **types différents** (chaînes de caractères, entiers, flottants, tableaux...) repérés par un **nom de champ**.

⇒ représentation de données composites et manipulation champ par champ ou globale (passage en argument des procédures simplifiées)

Structures **statiques** : champs de taille fixe

Structures **dynamiques** : comportant des champs de taille variable

On utilise une structure quand les objets manipulés ont **un lien entre eux** :

- différentes informations associées à une personne : nom, prénom, adresse, numéro de téléphone, ...
- différentes informations associées à un point du plan : nom, abscisse, ordonnée.
- échantillon de mesure sous ballon sonde : altitude, pression, température, humidité, vitesse, orientation du vent.

Écriture

```
int fputs(const char *s,
         FILE *stream)
```

permet d'écrire la chaîne `s` dans le flux `stream` (éventuellement `stdout` ou `stderr`).

La valeur de retour est non négative **quand tout s'est bien passé**, ou égale à `EOF` **en cas d'erreur**.

12.1.1 Exemple introductif de structures

```
#include <stdio.h>
#include <stdlib.h>
#define N 4 // nombre de points

// definition de la structure:
struct point{ // revient a definir un nouveau type
    int no;    // numero du point
    double x;  // abscisse du point
    double y;  // ordonnee du point
}; // ne pas oublier ;

int main(void){
    // declarations de variables de type struct point:
    struct point p; // variable ordinaire de type struct point
    struct point courbe[N]; // tableau d'elements de type struct point
```

```
// affectation de la structure:
for(int i=0; i<N; i++){
    p.no = i + 1;    // affectation du premier champ de p
    p.x = 1./(i+1.); // affectation du deuxieme champ de p
    p.y = 2.*i;     // affectation du troisieme champ de p
    courbe[i] = p;  // affectation de l'element i de courbe
}
// affichage des elements de la structure:
printf("Affichage des donnees:\n");
for(int i=0; i<N; i++){
    printf("%d %g %g\n", courbe[i].no, courbe[i].x, courbe[i].y);
}
exit(EXIT_SUCCESS);
}
```

MNI

256

2011-2012

MNI

257

2011-2012

12 Structures ou types dérivés

Langage C

12.1 Intérêt des structures

12 Structures ou types dérivés

Langage C

12.2 Définition, déclaration et initialisation des structures

Résultat à l'exécution :

```
1 1      0
2 0.5    2
3 0.333333 4
4 0.25   6
```

Caractéristiques de ce programme :

- définition d'une structure `point`,
- déclaration d'une variable du type `struct point` et d'un tableau dont chaque élément est du type `struct point`,
- affectation des structures champ par champ,
- affichage des résultats champ par champ.

12.2 Définition, déclaration et initialisation des structures

12.2.1 Définition d'un type structure point

Définir une structure correspond à définir **un nouveau type** :

```
struct point { // definition de la structure
    int no ;    // 1er champ
    double x ;  // 2eme champ
    double y ;  // 3eme champ
};
```

Important :

- Noter le `;` obligatoire après l'accolade `}`.
- La définition d'une structure ne réserve aucun emplacement mémoire associé à cette structure.

MNI

258

2011-2012

MNI

259

2011-2012

12.2.2 Quels types de champs peut-on mettre dans une structure ?

- **Tous les types de base du langage** (char, int, double...)
- **Des structures** (sauf du type en train d'être défini)
- **Des pointeurs** (y compris sur une structure du type en train d'être défini)
- **Des tableaux de taille fixe**
- **Des tableaux alloués dynamiquement** (sur le tas), via un pointeur
(⇒ **Tableaux automatiques déconseillés** : avertissement à la compilation)

12.2.3 Où placer la définition d'une structure ?

La définition de la structure doit être visible dans toutes les fonctions utilisant le type structure déclaré. On doit donc définir la structure en dehors de toute fonction (généralement en **début d'un fichier**), ou mieux dans un fichier include (d'extension .h)

12.3 Manipulation des structures

12.3.1 Accès aux champs d'une structure

Pour accéder aux champs d'une structure, on utilise l'opérateur `.` et le nom du champ :

```
debut.no = 1; // affectation du 1er champ
debut.x = 0.; // affectation du 2eme champ
debut.y = 0.; // affectation du 3eme champ
```

⇒ L'accès aux différents champs par leur nom est plus lourd que l'accès aux éléments d'un tableau (pas d'indice ⇒ pas de boucle). Ce nommage permet toutefois de rendre les programmes plus lisibles.

L'opérateur `.` est prioritaire par rapport à `&` (adresse) et `*` (indirection).

12.2.4 Déclaration de variables de type structure point

Déclarer de 2 variables de type **struct point** :

```
struct point debut, fin ;
```

Important : c'est la déclaration de variables du type **struct point**, ici **debut** et **fin**, qui réserve un emplacement mémoire de taille suffisante pour stocker tous les éléments associés à chaque variable.

12.2.5 Affectation d'une structure (constructeur)

Similaire au cas des tableaux :

- L'affectation globale peut se faire à la déclaration :

```
struct point fin={9,5.,2.};
```
- En dehors de la déclaration, l'affectation se fait champ par champ.

12.3.2 Affectation globale (même type)

Quand 2 structures sont du même type, **et uniquement dans ce cas**, on peut affecter globalement l'une à l'autre (copie)

```
fin = debut ;
```

Avantage sur les tableaux ! On dit que les structures sont des *lvalue*.

Par contre, **on ne peut pas** tester l'égalité (==) ou la différence (!=) entre 2 structures

```
if (fin == debut) {... est illégal !}
```

A fortiori, on ne peut pas utiliser les autres opérateurs relationnels (<, >...)

12.3.3 Entrées/sorties et structures

Obligatoirement champ par champ :

```
scanf("%d %g %g", &debut.no, &debut.x, &debut.y);
printf("%d %g %g", debut.no, debut.x, debut.y);
```

12.3.4 Retour sur l'exemple introductif

```
#include <stdio.h>
#include <stdlib.h>
#define N 4 // nombre de points

// definition de la structure:
struct point{ // revient a definir un nouveau type
    int no;    // numero du point
    double x;  // abscisse du point
    double y;  // ordonnee du point
}; // ne pas oublier ;

int main(void){
    // declarations de variables de type struct point:
    struct point p; // variable ordinaire de type struct point
    struct point courbe[N]; // tableau d'elements de type struct point
```

```
// affectation de la structure:
printf("Entrer les coordonnees des points:\n");
for(int i=0; i<N; i++){
    p.no = i + 1; // affectation du premier champ de p
    scanf("%lg %lg", &p.x, &p.y); // lecture des coordonnees
    courbe[i] = p; // affectation de l'element i de courbe
}
// affichage des elements de la structure:
printf("Affichage des donnees:\n");
for(int i=0; i<N; i++){
    printf("%d %g %g\n", courbe[i].no, courbe[i].x, courbe[i].y);
}
exit(EXIT_SUCCESS);
}
```

Résultat à l'exécution :

Entrer les coordonnees des points:

```
0. 1.
1. 1.
2. 1.
1.2 1.1
```

Affichage des donnees:

```
1 0 1
2 1 1
3 2 1
4 1.2 1.1
```

12.4 Représentation en mémoire des structures

Comme pour les tableaux : les champs des structures sont stockés dans l'ordre de leur déclaration et à proximité les uns des autres en mémoire.

Contrairement aux tableaux : il peut exister des « octets de remplissage » (*padding* en anglais) entre les différents champs, afin de respecter des contraintes d'alignement.

⇒ la taille (au sens de `sizeof`) d'une structure \geq la somme des tailles de ses différents champs.

```
struct mini_point{ // definition (aucune reservation memoire)
    float x; // 4 octets
    short no; // 2 octets
};
printf("%d", sizeof(struct mini_point));
```

Résultat 8 ⇒ **2 octets de remplissage** (sur une machine 32 bits) entre `no` et `x`

12.5 Pointeur sur une structure

On peut déclarer un pointeur sur une structure :

```
struct mini_point mpo, *pmpo=NULL; // declaration
```

```
pmpo = &mpo;
```

`pmpo` pointe alors sur le premier champ de la structure :

`pmpo` vaut `&mpo.no`

MNI

268

2011-2012

12.6.3 Structures contenant un tableau (taille variable)

Les tableaux automatiques (sur la **pile**) dans les structures provoquent des avertissements à la compilation \Rightarrow leur usage est déconseillé.

Utiliser des tableaux dynamiques (sur le **tas**) \Rightarrow pointeurs et `malloc` ou `calloc`:

```
struct courbe2{ // definition (aucune memoire reservee)
```

```
    int n;
```

```
    double *x;
```

```
    double *y;
```

```
};
```

```
...
```

```
struct courbe2 c; // declaration
```

```
scanf("d, &c.n); // nombre de points
```

```
c.x = (double *)calloc(c.n, sizeof(double)); // allocation
```

```
c.y = (double *)calloc(c.n, sizeof(double)); // sur le tas
```

```
c.x[0]=0.5;
```

```
c.y[0]=sqrt(c.x[0]);
```

MNI

270

2011-2012

12.6 Exemples de structures (plus ou moins) complexes

12.6.1 Tableaux de structures

```
struct point courbe[9] ; // declaration
```

```
courbe[0].x = 2. ;
```

abscisse du premier point de la courbe

12.6.2 Structures contenant un tableau (taille fixe)

```
struct courbe2{ // definition (aucune reservation memoire)
```

```
    double x[10];
```

```
    double y[10];
```

```
};
```

```
...
```

```
struct courbe2 c; // declaration
```

```
c.x[0]=0.5;
```

```
c.y[0]=sqrt(c.x[0]);
```

MNI

269

2011-2012

Attention, dans ce cas, aux **copies superficielles** : (copie du pointeur seulement)

```
struct courbe2 c1, c2; // declaration
```

```
...
```

```
c2 = c1; // copie superficielle
```

```
free(c1.x); // c2.x ne pointe plus vers une zone valide
```

Solution : la **copie profonde** (allouer un espace mémoire pour `c2.x` et `y` copier tous les éléments du tableau `c1.x`)

```
c2 = c1; // affectation des variables ordinaires
```

```
c2.x = (double *)calloc(c2.n, sizeof(double)); // allocation
```

```
for (int i=0; i < c2.n; i++) {
```

```
    c2.x[i] = c1.x[i]; // copie profonde
```

```
}
```

```
free(c1.x); // c2.x continue de pointer vers une zone valide
```

MNI

271

2011-2012

12.6.4 Structures contenant une structure

```

struct lieu{ // definition
    char nom[80];
    double longitude;
    double latitude;
};
struct observation{ // definition
    struct lieu ville; // imbrication
    double temperature;
};
...
struct observation obs; // declaration
obs.ville.longitude = 0.; // association de "."
obs.ville.latitude = 45.;
obs.temperature = 273.15;
printf(obs.ville.nom, "%s", "Bordeaux");

```

MNI

272

2011-2012

12.7 Structure et fonction, opérateur flèche

Les structures se comportent comme n'importe quel type du langage.

12.7.1 Passage par copie de valeur

C'est le mode par défaut : à utiliser **quand on ne veut pas modifier l'argument dans la fonction :**

```

void affiche_point(struct point p) {
    printf("%d %g %g\n", p.no, p.x, p.y); //affiche seulement
}
...
struct point po; // declaration
...
affiche(po); // passage par copie de valeur
...

```

MNI

274

2011-2012

12.6.5 Listes chaînées

Ce sont des structures contenant un pointeur vers une structure du même type

```

struct point { // definition
    int no ; //1er champ
    float x ; //2e champ
    float y ; //3e champ
    struct point *next ;
};

```

Elles permettent de mettre en œuvre des « structures » mathématiques de type graphe, arbre, etc.

MNI

273

2011-2012

12.7.2 Passage par copie d'adresse

À utiliser **quand on veut modifier l'argument dans la fonction :**

```

void init_point(struct point * p) {
    (*p).no = 1; // modification des valeurs
    (*p).x = 0.;
    (*p).y = 0.;
}
...
struct point po;
...
affiche(&po); // passage par copie d'adresse
...

```

MNI

275

2011-2012

12.7.3 Opérateur flèche

Afin d'éviter la notation lourde : `(*p_struct) . champ`

utiliser l'opérateur `->` (**défini uniquement pour les structures**) :

`p_struct -> champ`

On écrira la fonction précédente sous la forme :

```
void init_point(struct point * p) {
    p->no=1;
    p->x=0.;
    p->y=0.;
}
```

MNI

276

2011-2012

12 Structures ou types dérivés

Langage C

12.9 Exemple final

12.9 Exemple final

```
/* Fichier point.h */
#ifndef POINT
#define POINT
struct point { // définition du type point
    int no; /* numéro */
    float x; /* abscisse */
    float y; /* ordonnee */
};
#endif POINT /*POINT */
```

```
// programme sym3_pt.c
#include <stdio.h>
#include <stdlib.h>
#include "point.h"
```

MNI

278

2011-2012

12.8 Valeur de retour

Une fonction peut renvoyer une structure (elle est alors du type

`struct nom struct`):

```
struct point cree_point(int a, double b, double c) {
    struct point p; // declaration
    p.no = a;
    p.x = b;
    p.y = c;
    return p;
}
...
struct point po;
int n;
double x0, y0;
...
po = cree_point(n, x0, y0);
...

```

MNI

277

2011-2012

12 Structures ou types dérivés

Langage C

12.9 Exemple final

```
struct point psym(struct point m) {
// fonction à valeur de retour de type struct point
    struct point symetrique;
    symetrique.no = -m.no; // changement de signe
    symetrique.x = m.y; // échange entre x et y
    symetrique.y = m.x;
    return symetrique;
}

void sym(struct point m, struct point *n) {
// chgt de signe de no et échange x/y
    n->no = -m.no;
    n->x = m.y;
    n->y = m.x;
}
```

MNI

279

2011-2012

```

int main(void) {
    struct point a = {5, 1., -2.}; // déclaration de a
    struct point b, c; // déclaration de type struct point

    // affichage des champs de a
    printf("a = %d %g %g\n", a.no, a.x, a.y);

    // calcul et affichage de b
    b=psym(a);
    printf("psym(a) = %d %g %g\n", b.no, b.x, b.y);

    // calcul et affichage de c
    sym(a, &c);
    printf("sym. de a = %d %g %g\n", c.no, c.x, c.y);
    exit(EXIT_SUCCESS);
}

```

Résultat à l'exécution :

```

a = 5 1 -2
psym(a) = -5 -2 1
sym. de a = -5 -2 1

```

12.10 Bilan sur les structures

Les structures permettent de regrouper des données hétérogènes pour les communiquer de façon plus concise entre fonctions.

La notion de structure devient beaucoup plus puissante pour manipuler des objets complexes si on lui associe des **méthodes de manipulation** sous forme de

- fonctions (possible en C)
 - opérateurs (impossible en C mais possible en C++)
- ⇒ **programmation objet**

13 Éléments de compilation séparée

13.1 Introduction

Il est d'usage de **séparer les programmes « longs » en plusieurs fichiers** :

- il est plus facile et rapide de compiler séparément des **entités de programme courtes** que des centaines de lignes de code,
- la séparation en plusieurs fichiers permet de **structurer un programme**
⇒ fichier = unité cohérente contenant une ou quelques fonctions,
- le découpage en unités cohérentes permet une **ré-utilisation plus facile du code**,
- les différents fichiers peuvent être rassemblés dans des **bibliothèques** (collections de fichiers objets),
- il est possible d'automatiser la compilation séparée à l'aide de l'utilitaire **make**.

Le découpage en plusieurs fichiers induit des **contraintes**.

Il faut :

- permettre au compilateur de vérifier la cohérence entre définition/appels des fonctions : assurer la **visibilité des prototypes**,
- **accéder aux nouveaux types définis** (**struct . . .**) dans tous les fichiers où ils sont définis,

Dans les 2 cas, une mise en œuvre **robuste** de la compilation séparée réside dans l'utilisation de **fichiers d'entête** (*header files* en anglais).

13.2 Fichiers d'entête (*header files*)

13.2.1 Définition et usage

Ces fichiers peuvent contenir des déclarations de fonctions (**prototypes**) ou de nouveaux types.

Pour les fonctions, ils doivent être inclus dans :

- **le fichier où la fonction est définie**, pour assurer la cohérence déclaration/définition,
- **les fichiers où la fonction est appelée**, pour assurer la cohérence appel/déclaration.

L'inclusion se fait au moyen d'une directive préprocesseur : **#include**.

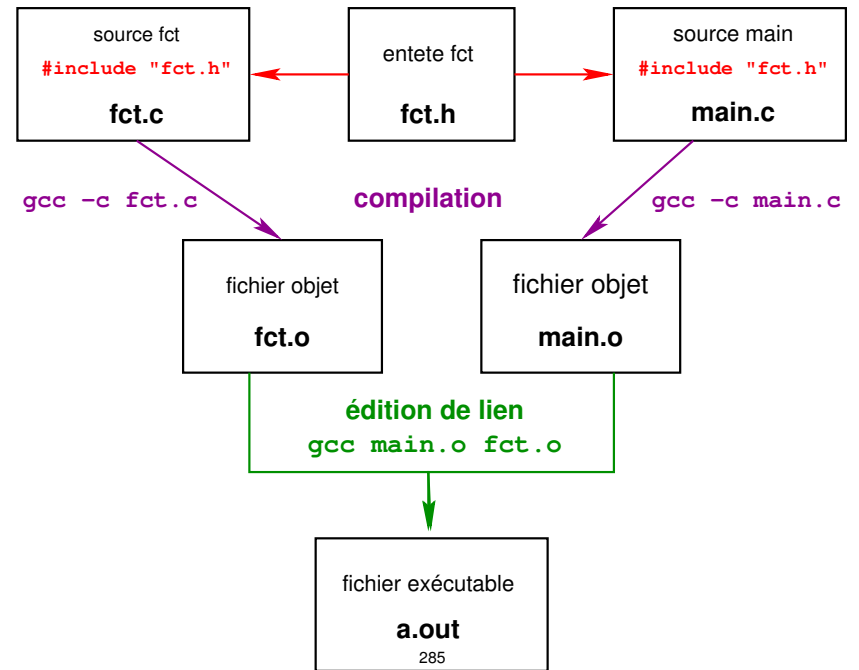
Pour les fonctions comme pour les types, il faut se protéger contre les **inclusions multiples**.

13.2.2 Structure d'un fichier d'entête

Soit un fichier `fct.c` contenant la définition de plusieurs fonctions. Le fichier `fct.h` correspondant peut s'écrire :

```
#ifndef FCT // si FCT n'est pas defini...
#define FCT // ... définir FCT...
double produit(double x, double y);
void affiche(double res);
#endif // ... fin du if
```

Dans l'exemple ci dessus, ce sont les directives préprocesseur qui protègent contre les **inclusions multiples**.



13.3 Exemple de programme en plusieurs fichiers

```
/* fichier main.c */
#include <stdio.h> // prototypes de printf, scanf...
#include <stdlib.h> // prototype de exit...
#include "fct.h" // prototypes de produit et affiche
int main(void) {
    double a, b, prod;
    printf("Entrer deux nombres reels:\n");
    scanf("%lg %lg", &a, &b);
    prod = produit(a,b); // appel de produit
    affiche(prod); // appel de affiche
    exit(EXIT_SUCCESS);
}
```



```

/* fichier fct.h : declaration des fonctions */
#ifndef FCT
#define FCT
double produit(double x, double y);
void affiche(double res);
#endif /* FCT */

```

```

/* fichier fct.c : definition des fonctions */
#include <stdio.h>
#include <stdlib.h>
#include "fct.h" // declaration de produit et affiche
double produit(double x, double y){
    return x*y;
}
void affiche(double res){
    printf("produit = %g\n", res);
}

```

13.4 Bibliothèques statiques de fichiers objets

- **Intérêt** : regrouper dans **un seul fichier** toute une collection de **fichiers objets** de fonctions compilées pour simplifier les futures commandes d'édition de lien qui utilisent ces fonctions.
- **Interface** : regrouper les prototypes de toutes les fonctions de la bibliothèque dans un seul fichier (.h).

13.4.1 Création et utilisation d'une bibliothèque statique (archive)

Pour créer une bibliothèque de nom **libtab** il faut :

1. compiler les fichiers à insérer dans la bibliothèque

```
gcc -c doubleld.c doubleld_libere.c
```
2. insérer les objets dans la bibliothèque (statique) :

```
ar rv libtab.a doubleld.o doubleld_libere.o
```

Compilation :

```
gcc-mni-c99 -c main.c fct.c
```

⇒ création de 2 fichiers objet : main.o et fct.o,

Édition de liens :

```
gcc-mni-c99 main.o fct.o -o main.x
```

⇒ création d'un fichier exécutable main.x

Caractéristiques de ce programme : 3 types de fichiers

- le fichier principal main.c : appel et déclaration de deux fonctions,
- le fichier fct.c : déclaration et définition de deux fonctions,
- le fichier d'entête fct.h : déclaration de deux fonctions définies dans fct.c et appelées dans main.c

3. créer un fichier **tab.h** contenant le prototype des fonctions utilisées
 ⇒ #include "**tab.h**" dans le code source utilisant la bibliothèque

4. puiser dans la bibliothèque lors de l'édition de lien :
 - en donnant le nom de l'archive pour un premier test :
(les fichiers sont alors tous dans le répertoire courant)

```
gcc main.c libtab.a
```

- ou en précisant le chemin d'accès :

```
gcc -I/home/user/include -L/home/user/lib main.c -ltab
```

où **tab.h** est dans le répertoire /home/user/include
 l'archive **libtab.a** est dans le répertoire /home/user/lib
 et les deux chemins dépendent de l'utilisateur de nom user

Remarque importante : les options **-I** et **-L** ne s'utilisent que pour une bibliothèque **non** standard.

Commande de gestion des bibliothèques statiques : **ar** (cf tar)

Actions principales :

r ajout ou remplacement d'une liste de membres

```
ar rv libtab.a doubleld.o doubleld_libere.o
```

t liste les membres de la bibliothèque

```
ar tv libtab.a
```

```
rw-r--r-- 904/800 860 Jan 29 11:17 2008 doubleld.o
```

```
rw-r--r-- 904/800 808 Jan 29 11:17 2008 doubleld_libere.o
```

x extraction d'une liste de membres

```
ar xv libtab.a doubleld.o
```

d destruction d'une liste de membres

```
ar dv libtab.a doubleld_libere.o
```

v option (verbose) avec messages d'information

u option (update) mise à jour seulement ⇒ **ar ruv**

13.4.2 Retour sur la bibliothèque standard

La bibliothèque standard est elle-même composée de sous-bibliothèques.
(les fichiers d'archive associés sont dans : /usr/lib ou /lib, ...)

A chaque sous-bibliothèque est associé un fichier d'entête :

(ces fichiers sont dans : /usr/include ou /include, ...)

– **stdio.h** : prototypes de scanf, printf, fopen, fclose, ...

– **stdlib.h** : prototype d'exit, définition de EXIT_SUCCESS, EXIT_FAILURE, ...

– **tgmath.h** (en C99) : prototype des fonctions mathématiques.

– ...

Dans le code source : le fichier d'entête est entre **<...>**

Édition de liens : automatique pour toutes les sous-bibliothèques sauf la sous-bibliothèque mathématique :

il faut utiliser l'option **-lm** de gcc.

13.4.3 Retour sur la bibliothèque libmnitab

Le fichier d'archive associé à cette bibliothèque est : **libmnitab.a**

Le prototype des fonctions de cette bibliothèque est dans le fichier **mnitab.h**

Dans le code source : `#include "mnitab.h"`

A la compilation : sur sappli utiliser `gcc+mni` ou `gcc+mni-c99`

⇒ ceci correspond à un alias vers :

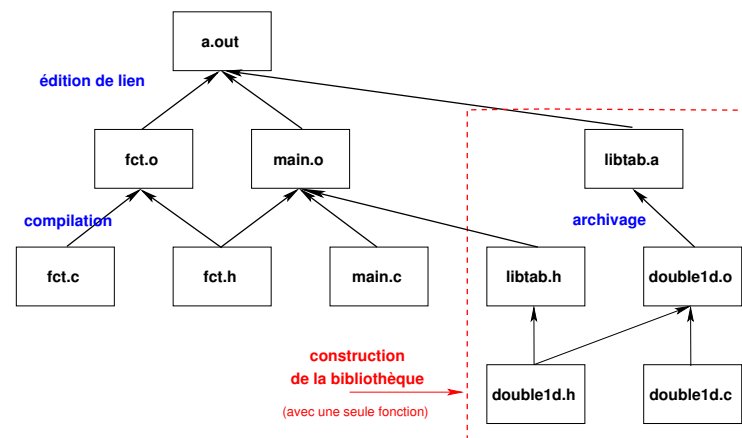
```
gcc -I/home/lefrere/include -L/home/lefrere/lib
```

où le répertoire /home/lefrere/include contient **mnitab.h**

et le répertoire /home/lefrere/lib contient **libmnitab.a**

Édition de liens : ajouter l'option **-lmnitab**

13.4.4 Bilan sur la création et l'usage d'une bibliothèque



Remarque : en général, la création d'une bibliothèque n'a d'intérêt que si elle contient de nombreuses fonctions que l'on n'a pas besoin de modifier. Sinon, préférer une compilation en fichiers séparés sans bibliothèque ou l'utilisation de l'utilitaire **make**.

13.5 Génération d'un fichier exécutable avec make

13.5.1 Principe

La commande **make** permet d'**automatiser** la génération d'un fichier exécutable ou **cible (target)** qui **dépend** d'autres fichiers en mettant en œuvre certaines **règles (rules)** de construction décrites dans un fichier **makefile**.

make minimise les opérations de mise à jour en s'appuyant sur les règles de dépendance et les **dates** de modification des fichiers.

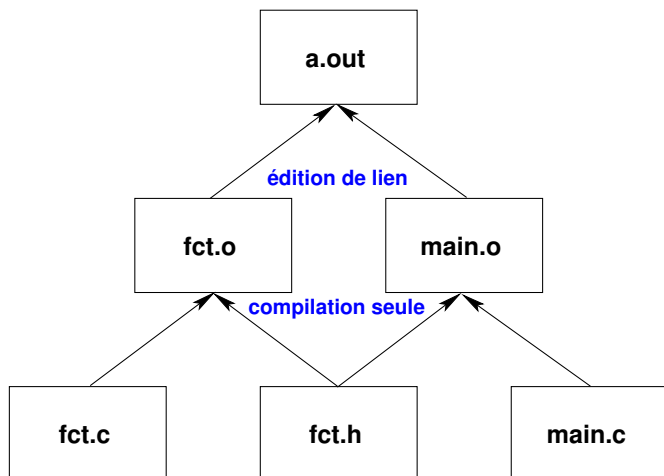
Application la plus classique :

reconstituer automatiquement un programme exécutable à partir des fichiers sources en ne recompilant que ceux qui ont été modifiés.

- **cible (target)** : en général un fichier à produire
- **règle** de production (rule) : liste des commandes à exécuter pour construire une cible (compilation pour les fichiers objets, édition de lien pour l'exécutable)
- **dépendance** : ensemble des fichiers nécessaires à la production d'une cible

13.5.3 Exemple élémentaire de makefile en C

Arbre des dépendances (exploré **récurivement** par make)



13.5.2 Construction d'un makefile

Le fichier **makefile** liste les cibles, décrit les dépendances et les règles.

Syntaxe des dépendances :

cible: liste des dépendances
(**tabulation**) règle de construction

⇒ nécessite d'intégrer des commandes shell dans le **makefile**

Le fichier **makefile** est construit à partir de l'arbre des dépendances.

gcc -MM fichier.c affiche les dépendances de **fichier.o** (nécessite les **.h**)

```
## fichier makefile construit a partir de l'arbre des depend
```

```
# première cible = exécutable => règle = édition de liens
```

```
a.out : fct.o main.o
__TAB__gcc fct.o main.o
```

```
# cibles des objets => règle = compilation seule avec gcc -c
```

```
fct.o : fct.c fct.h
__TAB__gcc -c fct.c
main.o : main.c fct.h
__TAB__gcc -c main.c
```

```
# ménage : suppression des fichiers restructuribles
```

```
clean:
__TAB__/_bin/rm -f a.out *.o
```

13.5.4 Utilisation d'un makefile

make *cible*

lance la production de la *cible* en exploitant le fichier *makefile* du répertoire courant.

make -n *cible*

affiche les commandes que devrait lancer *make* pour produire la *cible*

Remarque : *cmake* est un utilitaire équivalent à *make* et adapté à un ensemble plus large de systèmes d'exploitation (dont Windows).

Voir : <http://www.cmake.org/>

Annexe A : systèmes de numération

Système décimal

Représentation des nombres en base 10.

Les chiffres de la numération décimale sont les entiers de 0 à 9.

Exemple : $75000_{(10)} = 7 \times 10^4 + 5 \times 10^3 + 0 \times 10^2 + 0 \times 10^1 + 0 \times 10^0$

Système binaire

Représentation des nombres en base 2.

Les chiffres de la numération binaire sont le 0 et le 1.

Chaque chiffre correspond à un **bit**.

Exemples :

$$- 10_{(2)} = 1 \times 2^1 + 0 \times 2^0 = 2_{(10)}$$

$$- 1010_{(2)} = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 10_{(10)}$$

14 Conclusions

Le langage C est un **langage très complet** :

- à la fois haut niveau et bas niveau,
- applications : du numérique pour les physiciens à l'écriture de systèmes d'exploitation.

Une bonne connaissance du langage C permet d'aborder :

- des langages tels que le **Fortran** (FormulaTranslator) (passer de Fortran à C peut s'avérer plus compliqué que le contraire)
- des langages comme **PHP** et **java** dont la syntaxe est proche de celle du C,
- des langages orientés objet comme le **C++**,
- l'**interaction** entre langage et système d'exploitation (plus difficile en fortran) (commande de processus, acquisition de données, ...)

Passage d'une base b au système décimal

On note $r_{n-1} \dots r_1 r_0_{(b)}$, avec $0 \leq r_i < b$ ($i = 0, 1, \dots, n-1$), la représentation en base b d'un nombre composé de n chiffres.

La valeur décimale de ce nombre, notée p ou $p_{(10)}$, est alors donnée par :

$$p = \sum_{i=0}^{n-1} r_i \times b^i.$$

Application : $b = 2$, $n = 8$ (ensemble de 8 bits ou octet)

le cas où $r_0 = 0$, $r_1 = 1$, $r_2 = 0$, $r_3 = 1$ et $r_{i>3} = 0$

donne bien $00001010_{(2)} = 10_{(10)}$.

Passage du système décimal à une base b

Décomposer un nombre décimal, noté p ou $p_{(10)}$, en base b :

$$p_{(10)} = \sum_{i=0}^{n-1} r_i \times b^i = r_{n-1} \dots r_1 r_0 (b),$$

consiste à trouver le nombre de bits n et le poids affecté à chaque bit r_i .

Soient, $q_0 = p \div b$ et $q_{i+1} = q_i \div b$ quotient de la division entière de q_i par b .

On a alors :

- $r_i = q_i \% b$, le reste de la division entière de q_i par b .
- $n - 1$, l'indice du premier quotient nul : $q_{n-1} = 0$.

Application : soient $p = 6$ et $b = 2$. On a alors :

- $q_0 = 3, r_0 = 0$,
- $q_1 = 1, r_1 = 1$,
- $q_2 = 0, r_2 = 1 \Rightarrow n = 3$.

Donc : $6_{(10)} = 110_{(2)}$.

Tableau de passage entre système binaire et système décimal

décimal	binaire
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111

décimal	binaire
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111