UPMC

Master P&A/SDUEE

UE MP050

Méthodes Numériques et Informatiques

Langage C

2009–2010

Jacques.Lefrere@aero.jussieu.fr

albert.hertzog@lmd.polytechnique.fr

1 Introduction

1.1 Historique du C

– langage conçu dans les années 1970
– 1978 : parution de The C Programing Langage de B. KERNIGHAN et D. RICHIE
– développement lié à la diffusion du système UNIX
– 1988–90 : normalisation **C89** ANSI–ISO
  Deuxième édition du KERNIGHAN et RICHIE norme ANSI

– 1999 : norme **C99** en cours d'implémentation
  ajout de nouveaux types (booléen, complexe, entiers de diverses tailles (prise en compte des architectures 64 bits), caractères étendus (unicode), ...) ;
  introduction de la généricité dans les fonctions numériques,
  déclarations tardives des variables, tableaux automatiques de taille variable...
  $\Rightarrow$ se conformer à la norme ANSI pour la portabilité
– base d'autres langages dont C++ (premier standard en 1998), PHP, Java, etc.

1.2 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

1.3 Généralités

	langage C
format	«libre» $\Rightarrow$ mettre en évidence les structures par la mise en page (indentation)
ligne	pas une entité particulière (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	au plus 31 caractères alphanumériques plus _, commençant par une lettre ou _
maj/minuscule	distinction

1.4 Exemple de programme C avec une seule fonction

```c
#include <stdio.h>                          /* instructions préprocesseur */
#include <stdlib.h>

int main(void) {                            /* début du prog. principal */
  int i ;                                   /* déclaration */
  int s=0 ;                                 /* déclaration */

  for (i = 1 ; i <= 5 ;  i++) {             /* début de bloc */
      s += i ;
  }                                         /*   fin de bloc */
  printf("somme des entiers de 1 à 5\n");
  printf("somme = %d\n", s) ;

  exit(EXIT_SUCCESS) ;                      /* renvoie à unix le status 0 */
}                                           // fin du prog. principal
```

1.5 Structure générale d'un programme C

Programme C élémentaire

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

int main(void) {

Déclarations des variables;

Instructions exécutables;

}
```

- Un programme C = une (ou plusieurs) **fonction(s)** dont au moins la fonction `main` : le programme principal.

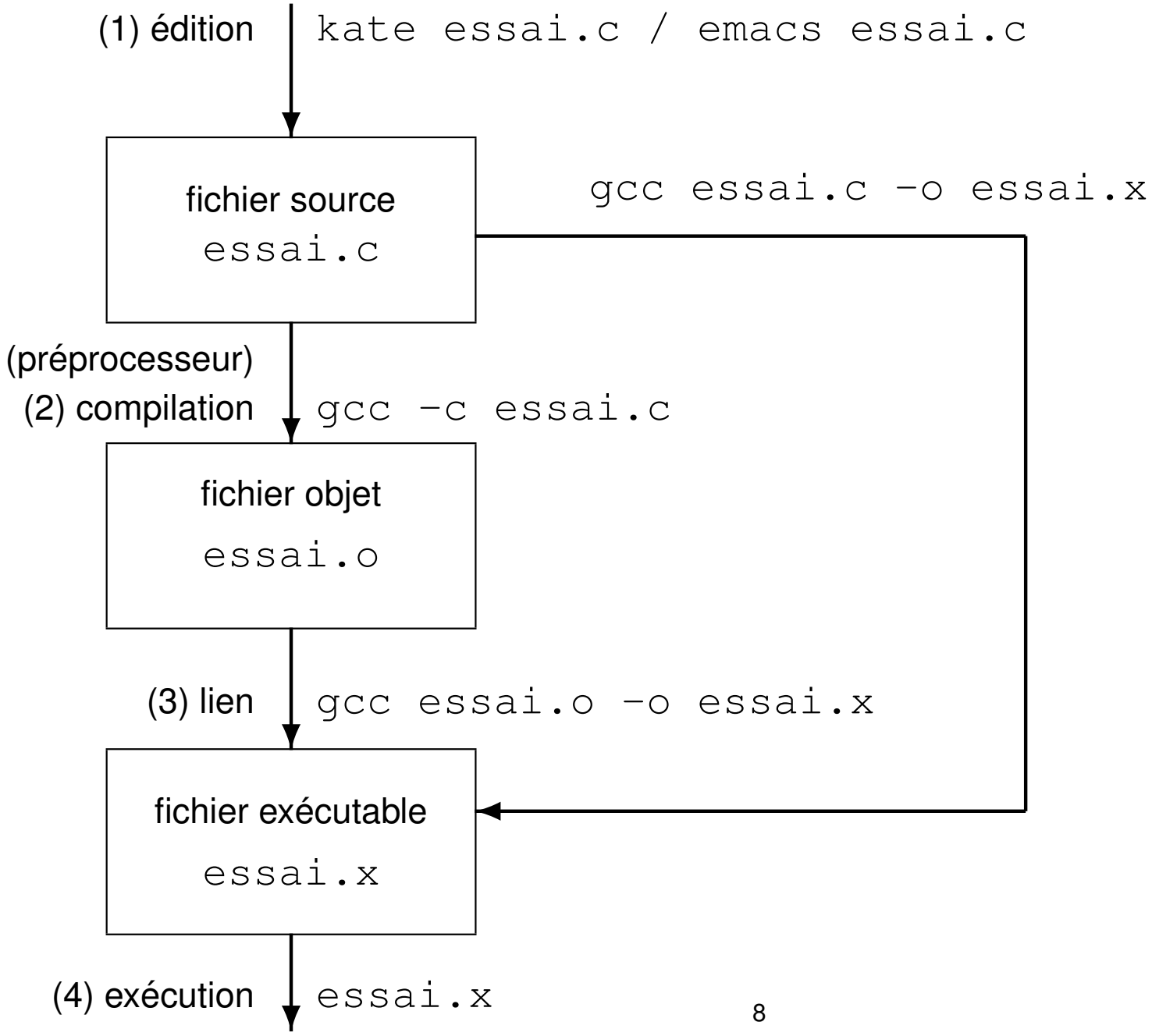
  N.-B : à l'extérieur de ces fonctions, il peut comporter des instructions, des déclarations de variables, des déclarations de fonctions spécifiant leur prototype, et des directives pour le préprocesseur introduites par `#`.
- Une instruction simple doit se terminer par `;`
- Une **instruction composée** est constituée d'un bloc d'instructions simples. Elle est délimitée par des accolades `{` et `}`.

1.6 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 #
2. **compilation** à proprement parler → fichier objet `.o`
3. **édition de lien** (`gcc` lance `ld`) → fichier exécutable `a.out`


(1) édition
kate essai.c / emacs essai.c
fichier source
essai.c
gcc essai.c -o essai.x
(préprocesseur)
(2) compilation
gcc -c essai.c
fichier objet
essai.o
(3) lien
gcc essai.o -o essai.x
fichier exécutable
essai.x
(4) exécution
essai.x

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

→ **`-l`**`truc` 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.

1.7 Compilateur

langage C
`gcc` (dans le shell)
avec options sévères
C89 (ANSI) $\longrightarrow$ alias **`gcc-mnic-89`**
C99$\longrightarrow$ 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 $\Rightarrow$ indiquer quelle est le type de la variable utilisée.

2.1 Types de base

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

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 types de base (dépend de la machine)

type		**`unsigned`**	**`signed`**
`char`	1 octet	$0 \rightarrow 255 = 2^8 - 1$	$-128 \rightarrow +127$
`short`	2 octets	$0 \rightarrow 2^{16} - 1$	$-2^{15} \rightarrow 2^{15} - 1$
`int`	4 octets	$0 \rightarrow 2^{32} - 1$	$-2^{31} \rightarrow 2^{31} - 1$
`long`	(4[a] ou) 8[b] octets	$0 \rightarrow 2^{64} - 1$	$-2^{63} \rightarrow 2^{63} - 1$

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

$\Rightarrow$ C99 : types entiers étendus à nb d'octets imposé ou à minimum imposé par exemple : **`int32_t`** ou **`int_least64_t`**

[a] Machine 32 bits

[b] Machine 64 bits

2.2 Valeurs maximales des entiers en C

```c
#include<stdio.h>
#include<stdlib.h>
#include<limits.h>
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) ;
}
```

2.2.1 Valeurs maximales des entiers en C : machine 32 bits

```
Unsig-Lng-Lng-max  18446744073709551615 ffffffffffffffff 1777777777777777777
Long-Long-max       9223372036854775807 7fffffffffffffff  777777777777777777
```

```
Unsigned-Long-max            4294967295         ffffffff           37777777777
Long-max                     2147483647         7fffffff           17777777777
Unsigned-Int-max             4294967295         ffffffff           37777777777
Int-max                      2147483647         7fffffff           17777777777
Unsigned-Short-max                65535             ffff               177777
Short-max                         32767             7fff                77777
```

2.2.2 Valeurs maximales des entiers en C : machine 64 bits

```
Unsig-Lng-Lng-max  18446744073709551615 ffffffffffffffff 1777777777777777777
Long-Long-max       9223372036854775807 7fffffffffffffff  777777777777777777
```

```
Unsigned-Long-max  18446744073709551615 ffffffffffffffff 1777777777777777777
Long-max            9223372036854775807 7fffffffffffffff  777777777777777777
Unsigned-Int-max             4294967295         ffffffff           37777777777
Int-max                      2147483647         7fffffff           17777777777
Unsigned-Short-max                65535             ffff               177777
Short-max                         32767             7fff                77777
```

2.3 Déclarations des variables

Indiquer le type de la variable + lui attribuer un identifiant (+ éventuellement l'initialiser)

C : en tête des fonctions

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

La déclaration d'une variable entraîne (généralement) la réservation d'un emplacement de la taille nécessaire pour stockée la variable dans la mémoire de l'ordinateur.

2.3.1 Syntaxe

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

Exemple de déclaration de 3 entiers : `int j, j2, k_max ;`

Exemple de déclaration avec initialisation : `int j = 2, a = 3 ;`

2.3.2 Valeurs

Type	langage C
`bool`	(C99) **`true`** **`false`**
`char`	`'a'`
Chaînes	`"chaine"` `"s'il"`
`short`	`17`**`s`**
`int` (décimal)	`17`
`int` (octal)	**`0`**`21` (attention)
`int` (hexadécimal)	**`0x`**`11`
`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`**

2.4 Constantes

2.4.1 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 $\Rightarrow$ on pourra également utiliser la directive **#define** du préprocesseur

```
#define J -1
```

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

2.4.2 Exemples d'attribut `const` en 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'
```

2.4.3 Exemples d'utilisation de `#define`

```
#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

3.1 Opérateur d'affectation

lvalue = expression $\Rightarrow$ conversions implicites par promotion de type

lvalue = (type) expression $\Rightarrow$ conversion forcée (opérateur **cast**)

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

```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
int a=123456789;
float b=0.123456789f, c;

printf("float : %d\nint   : %d\n", sizeof(float), sizeof(int)
c=(float)a;
printf("%d %.10g %.10g\n", a, c, b);
exit(EXIT_SUCCESS);
}
```

```
float : 4
int   : 4
123456789 123456792 0.123456791
```

3.2 Opérateurs algébriques

	langage C
addition	`+`
soustraction	`-`
multiplication	`*`
division	`/`
élévation à la puissance	≈ `pow(x,y)`
reste modulo	`%`

3.3 Opérateurs de comparaison

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

3.4 Opérateurs logiques

ET	&&
OU	||
NON	!

3.5 Incrémentation et décrémentation en C

- incrémentation
  - post-incrémentation : `i++` incrémente `i` d'une unité,
    après évaluation de l'expression
    `p=2; n=p++;` donne n=2 et p=3
  - pré-incrémentation : `++i` incrémente `i` d'une unité,
    avant évaluation de l'expression
    `p=2; n=++p;` donne n=3 et p=3
- décrémentation
  - post-décrémentation : `i--` décrémente `i` d'une unité,
    après évaluation de l'expression
    `p=2; n=p--;` donne n=2 et p=1
  - pré-décrémentation : `--i` décrémente `i` d'une unité,
    avant évaluation de l'expression
    `p=2; n=--p;` donne n=1 et p=1

3.6 Opérateurs d'affectation composée en C

lvalue opérateur = expression $\Rightarrow$ **lvalue = lvalue opérateur expression**

Exemples :

`j` **`+=`** `i`	$\Rightarrow$	`j = j + i`
`b` **`*=`** `a + c`	$\Rightarrow$	`b = b *` **`(`**`a + c`**`)`**

3.7 Opérateur d'alternative en C

exp1 ? exp2 : exp3 $\Rightarrow$ si **exp1** est vraie, **exp2**

sinon **exp3**

Exemple :

`c = (a>b)` **`?`** `a` **`:`** `b` affecte le max de `a` et `b` à `c`

3.8 Opérateurs agissant sur les bits

Le langage C possède des opérateurs de bas niveau travaillant directement sur les champs de bits.

	signification
~expr	négation
expr1 & expr2	et
expr1 | expr2	ou
expr1 ^ expr2	ou exclusif
expr1 << expr2	décalage à gauche de **expr2** bits
expr1 >> expr2	décalage à droite de **expr2** bits

3.9 Opérateur sizeof en C

Taille en octets d'un objet ou d'un type (résultat de type `size_t`).

Cet opérateur permet d'améliorer la portabilité des programmes.

`sizeof` (**identificateur**)

```
size_t n1; double a;
n1 = sizeof(a);
```

`sizeof`(**type**)

```
size_t n2;
n2 = sizeof(int);
```

3.10 Opérateur séquentiel , en C

expr1 , expr2 permet d'évaluer successivement les expressions **expr1** et **expr2**.

Utilisé essentiellement dans les structures de contrôle (`if`, `for`, `while`).

3.11 Opérateurs & et * en C

&objet $\Rightarrow$ adresse de l'objet

***pointeur** $\Rightarrow$ valeur pointée (indirection)

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 `,`

$\Rightarrow$ indiquer les priorités avec des parenthèses !

4 Introduction à `printf` et `scanf`

4.1 `printf` et `scanf`

`printf` permet d'afficher à l'écran (`stdout`) des messages et les valeurs des variables

`scanf` de lire du clavier (`stdin`) les valeurs des variables

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

int main(void) {
int i;
double x;

printf("Entrer un entier\n");
scanf("%d", &i);
printf("La valeur de i est %d\n", i);
printf("Entrer un réel\n");
scanf("%lf", &x);
printf("Les valeurs de i et x sont %d et %g\n", i, x);

exit(EXIT_SUCCESS);
}
```

4.2 Formats

4.2.1 `printf`

Type	Format
`char`	`%c`
`int/short`	`%d`
`unsigned int/unsigned short`	`%ud` (`%o`, `%x`)
`long`	`%ld`
`unsigned long`	`%lu` (`%lo`, `%lx`)
`long long`	`%lld`
`unsigned long long`	`%llu`
`float/double`	(`%e`, `%f`) `%g`
`long double`	(`%le`, `%lf`) `%lg`

4.2.2 scanf

Type	Format
char	%c
short	%hd
unsigned short	%hu(%ho, %hx)
int	%d
unsigned int	%u(%o, %x)
long	%ld
unsigned 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

5.1 Structure `if`

permet de **choisir** quelles instructions vont être exécutées

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

```
if (expression) {
  bloc d'instructions;
  \\ si l'expression est vraie
} else {
  bloc d'instructions;
  \\ si l'expression est fausse
}
```

L'expression est vraie si elle est `!=0`, et fausse si elle est `==0`.

Exemples de `if`

```
#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);
}
```

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.

5.2 Structure `switch` (pas avec des flottants)

permet de programmer des choix 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.

Exemples de case

```
#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;      /* necessaire ici ! */
```

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

```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 ';' :
  case ':' :
```

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

5.3 Structures itératives et branchements

Les structures itératives (ou boucles) permettent de répéter plusieurs fois un bloc d'instructions.

5.3.1 Boucle définie

Quand le nombre de répétitions est connue, on utilise **for**

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)

Exemples de for

```
#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);
}
```

5.3.2 Boucle indéfinie

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

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

Les instructions sont répétées ”tant que” expr est vraie dans le premier cas.

Les instructions sont exécutées au moins une fois, puis répétées ”tant que” expr est vraie dans le second cas.

NB : **Attention** aux boucles infinies si expr est toujours vrai !

Exemple de `while`

```
#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) ;
}
```

5.3.3 Branchements

Les branchements permettent de modifier le comportement des boucles.

bouclage anticipé	`continue;`
sortie anticipée	`break;`
branchement	`goto étiquette;`

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

Exemple de continue

```
#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) ;
}
```

Exemple de break

```
#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 Fonctions en C

6.1 Généralités

Une fonction sert à :

– rendre un programme plus lisible ;

– factoriser un nombre d'opérations (ex : calculs).

En règle générale, une fonction admet des arguments et renvoie une valeur (de retour). La fonction C est alors très proche d'une fonction mathématique. Le type de la valeur de retour définit le type de la fonction (qui peut être n'importe quel type du langage, ou une structure). La liste des arguments ou la valeur de retour (voire les deux) peut également être vide (`void`).

6.2 Définition d'une fonction

6.2.1 Fonctions retournant une valeur

```
type fct(type x1, ...)
/* déclaration des arg. dans l'entête */
{
/* corps de la fonction */
    type varloc ; /* variable locale à la fonction*/
...
    return expression ; /* du type de la fonction */
}
```

Exemples de fonction retournant une valeur

```
int som(const int p) {
/* somme des p premiers entiers */
  int i , s;  /* var. locales  */
  s = 0;
  for (i = 0; i <= p; i++){
      s += i;
  }
  return s ;  /* valeur rendue  */
}
```

6.2.2 Fonctions C sans retour

```
void fct(type x1, ...)
/* déclaration des arg. dans l'entête */
{
/* corps de la fonction */
    type varloc ; /* variable locale */
...
    return ;
}
```

6.2.3 Appel de fonction

fonctions typées avec retour **return** expression **;**	calcul de la valeur de retour dans la fonction
`printf("%d", fct(x1, x2, ...))` `y=2*fct(x1, x2, ...);`	utilisables dans des expressions dans l'appelant
fonctions de type **void** **return ;** (sans expression)	procédures à effets de bord ou modif des paramètres
`fct(x1, x2, ...) ;`	invocation dans l'appelant

6.3 Visibilité des fonctions (au sein d'un fichier)

Une fonction doit être délarée pour permettre au compilateur de vérifier :

– la concordance entre le nombre d'arguments dans l'appel et dans la définition c
– pour permettre de mettre en place des conversions implicites (pour les arguments et la valeur de retour) entre l'appel et la définition

6.3.1 1ière méthode de déclaration : la définition

La définition tient lieu de déclaration ⇒ définir la fonction avant de l'appeler.
Pb : cette méthode est vite compliquée car une fonction peut en appeller d'autres qui en appellent d'autres, etc.

6.3.2 2ième méthode de déclaration : le prototype

Le prototype de la fonction définit son type, son nom, et le nombre et le type de ses arguments.

```
type fct(type arg1, ..., type argN);
```

Ce prototype peut être placé (par ordre croissant de préférence) :

- au sein d'une fonction ⇒ visible uniquement dans cette fonction
- au début du fichier (en dehors de toute fonction) ⇒ visible dans tout le fichier
- dans un fichier d'entête (ou « include »), d'extension `.h`, inclus par

  `#include "fichier.h"`

  ⇒ visible dans tout les fichiers qui incluent ce fichier `.h`

6.4 La fonction principale

main est la fonction principale d'un programme C : celle par où le programme débute

```
int main(void)
```

main est de type **int** ⇒ `exit(EXIT_SUCCESS);` renvoie au shell la valeur `EXIT_SUCCESS` à la fin

main ne prend pas d'arguments (de la ligne de commande shell)

Sont généralement inclus dans le fichier par des directives **#include** :

<stdio.h> qui contient les prototypes de `printf` et `scanf`

<stdlib.h> qui contient le prototype de `exit`

et

<tgmath.h> qui contient les prototypes des fonctions mathématiques

6.5 Passage des arguments dans la fonction

déclarer types dans l'entête	arguments
conversion implicite	entre définition (argument formel) et appel (arg. effectif)
par **valeur** (copie locale) $\Rightarrow$ pas de modification possible au retour dans l'appelant	passage d'arguments
par valeur + `const`	pour permettre au compilateur de vérifier que l'argument n'est pas modifié

6.6 Exemple

6.6.1 Faux échange en C

```
/* Fonctions : passage des arguments par valeur */
/* => pas de modification en retour */
#include<stdio.h>
#include<stdlib.h>

void echange(int a, int b)
{
int c;  /* variable locale à la fonction */
printf("\tdebut echange : %d %d \n", a, b);
c = a;
a = b;
b = c;
printf("\tfin echange : %d %d \n", a, b);
return;
}
```

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

En fait, la fonction `echange` travaille sur une **copie des variables** `a` et `b`.

```
avant appel : n=1 p=5
        debut echange : 1 5
        fin echange : 5 1
apres appel : n=1 p=5
```

6.7 Variables locales aux fonctions

Les variables **locales** aux fonctions sont des variables **temporaires** ou **automatiques** : leur emplacement en mémoire est alloué lors de l'appel de la fonction et (automatiquement) libéré au retour.
On dit que leur portée est réduite à la fonction.

7 Introduction aux pointeurs

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

7.1 Déclaration

Rappel : déclaration d'une variable

```
int a;
```

= réservation d'une zone mémoire dont la taille dépend du type.

Déclaration d'un pointeur

```
int *ptr;
```

= réservation d'une zone mémoire pour stocker **l'adresse (et la taille)** d'une variable du type pointée.

7.2 Affectation

Rappel : Affecter une valeur à une variable

`a=3;`

= stocker la valeur dans la zone mémoire réservée lors de la déclaration de la variable.

Affecter une adresse à un pointeur

`ptr=&a;`

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

On dit que (le pointeur) `p` pointe vers (la variable) `a`.

Attention : de même que la valeur d'une variable est indéterminée avant son initialisation ($= 1^{\text{ière affectation}}$), de même l'adresse contenue dans le pointeur est indéterminée avant l'initialisation du pointeur $\Rightarrow$ **Initialiser un pointeur avant de le manipuler.**

7.3 Indirection (opérateur $*$)

L'indirection est l'opération permettant d'accéder au contenu de la variable pointée via le pointeur.

```
*ptr=4;
```

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

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

La déclaration d'un pointeur :

```
double *pt_b;
```

peut se lire **`*pt_b`** est un **`double`**

7.4 Dissociation

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

```
*ptr=NULL;
```

7.5 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
```

7.5.1 Vrai échange en C

```c
/* Fonctions : passage des adresses en arguments (par valeur
/* =>  modification en retour */
#include<stdio.h>
#include<stdlib.h>

void echange(int *pa, int *pb){
/*      pa et pb  ^          ^  pointeurs sur des entiers */
int c;  /* variable locale à la fonction  (pas pointeur) */
printf("\tdebut echange : %d %d \n", *pa, *pb);
printf("\tadresses debut echange : %p %p \n", pa, pb);
c = *pa;
*pa = *pb;
*pb = c;
printf("\tfin echange : %d %d \n", *pa, *pb);
```

```
printf("\tadresses fin echange : %p %p \n", pa, pb);
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 adresses */
printf("apres appel : n=%d p=%d \n", n, p);
printf("adresses après appel : %p %p \n", &n, &p);
exit(0) ;
}
```

Ici, la fonction `echange` travaille sur une **copie de l'adresse des variables** `a` et `b`, ce qui permet de modifier les variables via l'opérateur d'indirection.

```
avant appel : n=1 p=5
adresses avant appel : 0xbffbf524 0xbffbf520
        debut echange : 1 5
        adresses debut echange : 0xbffbf524 0xbffbf520
        fin echange : 5 1
        adresses fin echange : 0xbffbf524 0xbffbf520
apres appel : n=5 p=1
adresses après appel : 0xbffbf524 0xbffbf520
```

7.6 Retour sur printf/scanf

`printf` ne modifie pas les variables qu'il reçoit en argument ⇒ passage par valeur

```
printf("x=%g\n",x);
```

`scanf` modifie la valeur des variables qu'il reçoit en argument ⇒ passage par adresse (ou par référence)

```
scanf("%g",&x);
```

8 Tableaux

8.1 Définition et usage

Un **tableau** est un ensemble d'objets **de même type**. L'ensemble des objets est identifié par un **identifiant unique**.

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.

8.2 Tableaux de dimension fixe

Déclaration

Tableau à 1 dimension (vecteur) :

```
int tab1[3] ;
```

Tableau à 2 dimensions (matrice) :

```
double tab2[3][2] ;
```

etc...

Référence à un élément

1D : `i = tab1[2] ;`

2D : `i = tab2[2][0];`

♠ opérateur séquentiel « `,` » $\Rightarrow$ `[2,0]` interprété comme `[0]`

Indexation

commence à **0** (!) $\Rightarrow$ termine à `N-1`, où N est le nombre d'éléments du tableau.

`tab[2]` est le **troisième** élément du tableau.

Attention : généralement, aucun contrôle sur la valeur de l'indice n'est effectué par le compilateur $\Rightarrow$ risque important de `segmentation fault` (voir plus loin)

Modification des éléments d'un tableau

À la déclaration :

Vecteur :

```
int tab1d[3]={1,2,3};
```

produit le tableau `tab1d` tel que `tab1d[0]=1`, `tab1d[1]=2`, etc.

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.

Hors de la déclaration, il est impossible de préciser les valeurs d'un tableau en une seule instruction $\Rightarrow$ recours aux boucles :

```
for (i=0; i<3; i++) {
   tab1d[i]=i+1;
}
```

produit le tableau `tab1d` précédent

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

produit le tableau `tab2d` précédent

Rangement des éléments en mémoire

2 règles à retenir :

1) Les éléments d'un tableau sont rangés **de manière contiguë** en mémoire

2) Dans le cas de tableaux à plusieurs dimensions, l'indice qui défile le plus vite (celui des éléments contigus) est **l'indice le plus à droite** ($\Rightarrow$ conséquence importante sur les performances du programme)

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

1er indice = lignes de la matrice

2e indice = colonnes de la matrice

8.3 Inconvénients des tableaux de dimensions fixes

Très fréquemment, on est amené à faire tourner un même programme sur un ensemble d'éléments qui **varie d'une exécution à l'autre** (par ex : résolution avec laquelle on examine une fonction) $\Rightarrow$ l'utilisation des tableaux de taille fixe (tel que ci-dessus) nécessite une modification de nombreuses lignes de code (risque d'erreurs)

1ère (ancienne) solution : utiliser une directive du préprocesseur pour paramétrer la taille

```
#define N 20
```
(noter l'utilisation de la majuscule)

```
int main(void) {
double tab[N];
...
for (int i=0; i<N; i++) {
...
```

Changer la taille du tableau ne nécessite alors que la modification de la 1ère ligne

2e (nouvelle) solution : en **C99**, on peut profiter de la possibilité offerte par les déclarations tardives

`scanf("%d", &n);` (on demande à l'utilisateur la taille)

`double tab[n];` (tableau de"taille variable")

NB : le tableau `tab` est déclaré **après** une instruction.

Il n'y a plus rien à modifier (! ! !), le code est compilé une fois pour toutes.

Attention : la valeur de `n` doit être raisonnable avant la déclaration de `tab` $\Rightarrow$ le code suivant compile, mais a de grandes chances de provoquer une erreur

`int n;` (La valeur de `n` est indéfinie ici ! !)

`double tab[n];`

Exemple de programme utilisant les déclarations tardives

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

int main(void) {
int n; // taille du vecteur

printf("Entrer la taille du vecteur\n");
scanf("%d", &n); // taille fixee par l'utilisateur
// tableau de taille variable => decl. tardive
// on peut utiliser tab partout sous sa declaration
double tab[n];
printf("*** Saisie des elements ***\n");
for (int i=0; i<n; i++) { // decl. tardive de i
   // (utilisable uniquement dans la boucle)
   printf("tab[%d] ? ", i);
   scanf("%lg", &tab[i]);
}
```

```
printf("*** Affichage des elements ***\n");
for (int i=0; i<n; i++) { // decl. tardive de i
   // 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] ? 2.5
tab[1] ? -4.45
tab[2] ? 0.123
*** Affichage des elements ***
tab[0] = 2.5
tab[1] = -4.45
tab[2] = 0.123
```

8.4 Tableaux et pointeurs en C

En langage C, tout **identifiant de tableau** est converti en un **pointeur constant sur le type des éléments** du tableau, dont la valeur est l'**adresse du premier élément** du tableau :

`float tab[9] ;` $\Rightarrow$ `tab` est converti en un pointeur vers `tab[0]`

$\Rightarrow$ **`tab = &tab[0]`** $\Rightarrow$ **`tab[0] = *tab`**

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

l'addition `tab+i` d'un entier `i` au pointeur `tab` est interprétée comme l'adresse de l'élément d'indice `i` du tableau :

`tab[i]` <=> **`*(tab+i)`** et **`&tab[i]`** <=> **`tab+i`**.

Par exemple, `pf` sous-tableau commençant au 4^{e} élément de `tab` :

`float *pf ;  pf = &tab[3] ;` ou aussi `pf = tab + 3 ;`

8.4.1 Ordre des éléments de tableaux 2D en C

```
#include <stdio.h>
#include <stdlib.h>
#define LIGNES 3
#define COLONNES 5

/* impression d'un tableau à 2 dimensions */
int main(void) {
int t [LIGNES] [COLONNES] = { {11,12,13,14,15},
                              {21,22,23,24,25},
                              {31,32,33,34,35}
                            } ;
int i, j;
int *k = &t[0][0];
printf("impression du tableau 2d t[i][j]=10i+j+1\n");
printf("en faisant varier j à i fixé\n");
```

```
for (i = 0 ; i < LIGNES; i++) {
    for (j = 0 ; j < COLONNES; j++) {
       printf("%d ", t[i][j]) ;
    }
    printf("\n") ;
}
printf("impression du tableau 2d t[i][j]=10i+j+1\n");
printf("en suivant l'ordre en mémoire\n");
for (i = 0 ; i < LIGNES * COLONNES; i++) {
    printf("%d ", *(k+i)) ; // ou meme k[i]
}
printf("\n") ;
printf("=> l'indice le plus à droite varie le plus vite\n");
exit(0) ;
}
```

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

$\Rightarrow$ l'indice le plus à droite varie le plus vite (rangement par lignes)

Sous-tableau 1D avec un pointeur

```c
#include <stdio.h>
#include <stdlib.h>
#define N 10

/* manipulation d'un sous-tableau 1d avec un pointeur */
int main(void) {
double tab[N]; /* tableau initial */
double *ptrd=NULL; // pointeur sur le même type
                   // que les élémts du tableau
int i;

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

```
}
ptrd = tab + 3; /* arithm.pointeurs: 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) ;
}
```

```
 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
```

8.5 Fonctions et tableaux

Passage de tableaux de dimensions fixes (pour le compilateur)

La dimension du tableau est une expression constante qui est soit codée en dur au sein du programme (à éviter), soit spécifiée par une directive `#define` du préprocesseur $\Rightarrow$ pas de problème particulier

```
#include <stdio.h>
#include <stdlib.h>
#define N 4
#define P 8
void init_et_print(int t[N][P]) { // taille fixe
for (int i=0; i < N; i++) {
   for (int j=0; j < P; j++) {
      t[i][j]=i-j;              // initialisation
      printf("%2d ",t[i][j]); // affichage
   }
   printf("\n");
}
}
```

```
int main(void) {
int tab[N][P]; // taille fixe

init_et_print(tab); // en fait, un pointeur est transmis
printf("dans main : tab[0][4]=%d\n",tab[0][4]);
exit(EXIT_SUCCESS);
}
```

```
 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 : tab[0][4]=-4
```

pointeur $\Rightarrow$ valeurs modifiables

problème : la fonction `init_et_print` ne peut être utilisée que pour des tableaux de 4 lignes et 8 colonnes (à cause des tailles fixes) $\Rightarrow$ intérêt limité

Passage de tableau de dimension inconnue à la compilation

Transmettre le nombre d'éléments à la fonction et (C99) utiliser les tableaux de taille variable

```
#include <stdio.h>
#include <stdlib.h>
void init_et_print(const int n, const int p, int t[n][p]) {
// t est 1 tableau de taille variable
// ATTENTION : n et p à déclarer avant t dans la liste d'arguments
for (int i=0; i < n; i++) {
   for (int j=0; j < p; j++) {
      t[i][j]=i-j;              // initialisation
      printf("%2d ",t[i][j]); // affichage
   }
   printf("\n");
}
}
```

```c
int main(void) {
int n, p;
printf("Entrer n et p\n");
scanf("%d %d", &n, &p);
int tab[n][p]; // taille variable, definie par l'utilisateur

init_et_print(n,p,tab); // en fait, un pointeur est transmis
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
```

Ici, la fonction `init_et_print` fonctionne pour tous les tableaux à deux dimensions, quelles que soient les tailles des 2 dimensions.

Limite des tableaux de taille variable

Les tableaux de taille variable sont la solution à utiliser dans la plupart des cas

...Mais, comme toutes les variables considérées jusqu'ici, les tableaux de taille variable sont visibles dans le programme :

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

En particulier, les tableaux de taille variable, s'ils sont déclarés dans une fonction, ne sont pas visibles dans la fonction appelante.

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

```c
#include <stdio.h>
#include <stdlib.h>
void print_1D(const int n, const int t[n]) {
  for (int i = 0; i < n; i++) {
    printf("%d ", t[i]) ;
  }
  printf("\n");
}
void tab_var(const int nn){
  int tab[nn]; // tab 1D local, declaré sur la pile
  for (int i=0; i<nn; i++) {
     tab[i] = i;
  }
  print_1D(nn, tab);
}
```

```c
int main(void) {
  int n ;
  printf("entrer n ");
  scanf("%d", &n);
  tab_var(n);
  // tab n'est pas visible dans le main
  exit(0) ;
}
```

```
entrer n 3
0 1 2
```

Le programme modifié ne fonctionne pas non plus

```
...
int * tab_var(const int nn){
  int tab[nn]; // tab 1D local, declaré sur la pile
  for (int i=0; i<nn; i++) {
     tab[i] = i;
  }
  print_1D(nn, tab);
  return tab;
}
int main(void) {
  int n, *t;
  printf("entrer n ");
  scanf("%d", &n);
  t=tab_var(n);
  printf("%d \n",t[1]);
  exit(0) ;
}
```

Avertissement à la compilation

```
gcc -std=c99 -W -Wall -pedantic tab-pile2.c
tab-pile2.c: Dans la fonction « tab_var »:
tab-pile2.c:15: AVERTISSEMENT: fonction retourne l'adresse
d'une variable locale
```

Et problème (aléatoire) à l'exécution

```
entrer n 1
0
1108562408
```

Ceci est un problème dans le cas suivant :

– un fichier contient les dimensions d'un tableau, et les valeurs des éléments
– on veut écrire une fonction qui lit ce fichier, « alloue » le tableau, et le remplit
– cette fonction est appelée (par exemple) dans `main`

Comme dans le cas précédent, le tableau n'est pas visible dans `main`, alors que l'on voudrait l'utiliser pour faire des calculs

$\Rightarrow$ Dans ce cas, la solution est d'utiliser une autre partie de la mémoire de l'ordinateur (le « tas ») qui est géré manuellement par le programmeur. Le tas permet d'assurer une permanence des variables.

9 Allocation dynamique (sur le tas)

Allocation

2 fonctions permettent d'allouer un espace mémoire sur le tas :

`void *malloc(size_t size)` : allocation de `size` octets

Exemple

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

réalise l'allocation d'un tableau de 10 `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`

`void *calloc(size_t nmemb, size_t size)` réalise l'allocation de `nmenb*size` et initialise l'espace alloué à **0**

Exemple

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

⇒ `ptr[0]=0...ptr[9]=0`

Sinon utilisation similaire à `malloc`

En cas d'échec de l'allocation

les 2 fonctions retournent le pointeur **`NULL`**

Libération de la mémoire allouée

Rappel : la gestion du « tas » est manuelle : la libération de l'espace alloué est à faire par le programmateur par la fonction `free`

```
void free(void *ptr)
```

où `ptr` est un pointeur initialisé par `malloc` ou `calloc`

ajouter `ptr=NULL;` pour plus de sécurité.

Attention aux fuites de mémoire

```
int *p=NULL, a=1;
p=(int *)calloc(100,sizeof(int));
p=&a;
```

Plus aucun pointeur ne pointe vers l'espace alloué par `calloc` $\Rightarrow$ cette partie de la mémoire est perdue pour le restant du programme, il est impossible d'y accéder à nouveau.

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

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

9.1 Allocation d'un tableau 1D

```c
#include <stdio.h>
#include <stdlib.h>
int * alloc_et_init(const int n) {
   int *p=NULL;

   p=(int *)calloc(n, sizeof(int));
   if (p == NULL) {
     printf("Erreur d'allocation\n");
     return p; // retour dans main avec le pointeur nul
   }
   for (int i=0; i < n; i++) {
     p[i]=i;
   }
   return p; // !! p est déclaré sur le tas
}
```

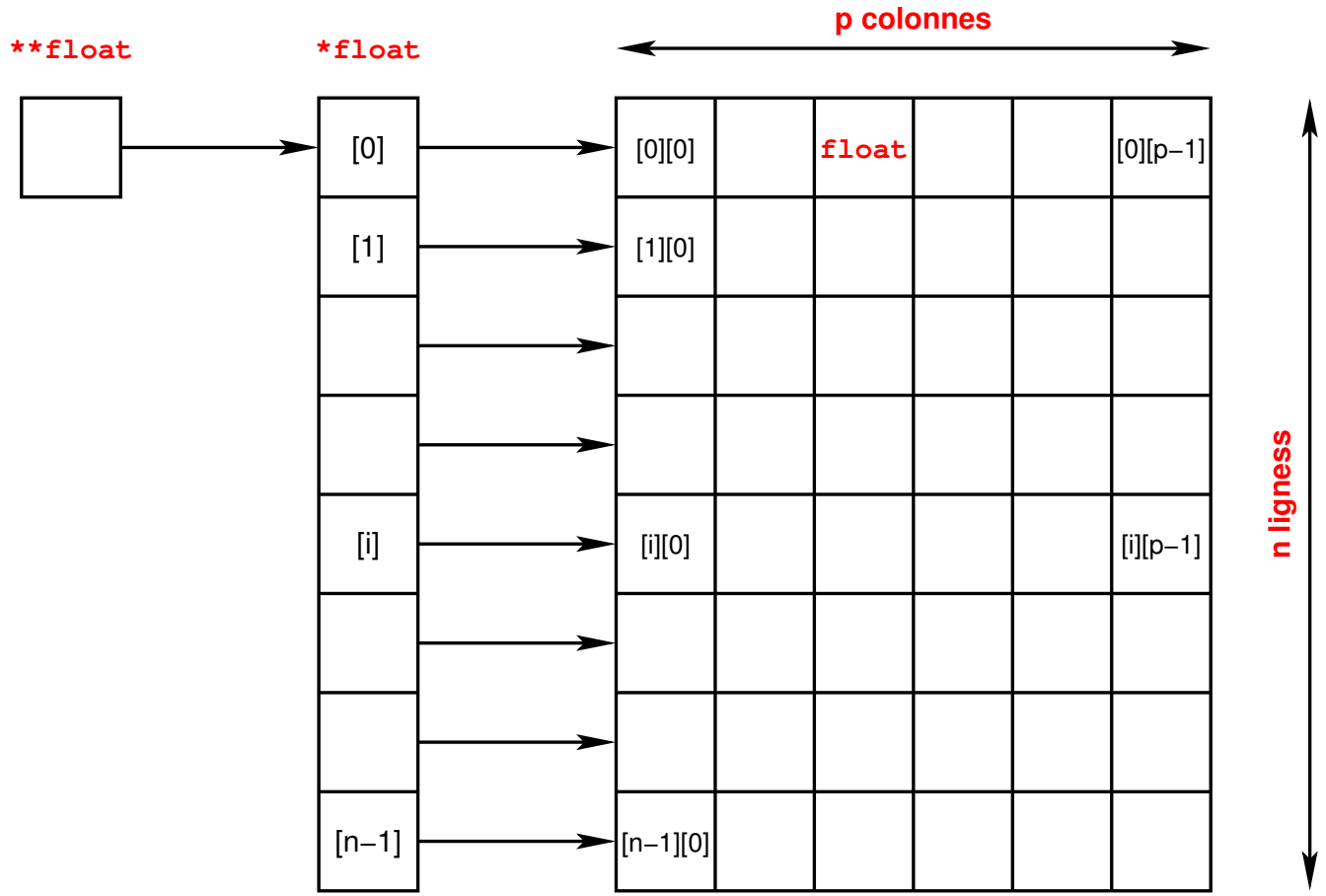
```c
int main(void) {
   int n, *pti = NULL; /* initialisation à NULL */

   printf("Entrer n ");
   scanf("%d",&n);
   pti=alloc_et_init(n);
   if (pti != NULL) { // allocation OK
     for (int i=0; i < n; i++) {
       printf("%d ",pti[i]);
     }
   }
   printf("\n");
   free(pti); // libération de l'espace
   pti=NULL; // sécurité
   exit(0);
}
```

```
Entrer n 6
0 1 2 3 4 5
```

9.2 Matrices de taille quelconque

Allocation dynamique d'un tableau 2D en C $\Rightarrow$ pointeur de pointeurs



```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;
   }
```

```
// 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++) {
       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
```

9.3 Résumé

1. 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]`)
2. si le tableau est utilisé dans la fonction appelant la fonction où le tableau est défini, il faut utiliser les allocations dynamiques sur le tas (`int **tab`)

Problème : les deux types de déclarations ne sont pas équivalentes, et l'on ne peut pas utiliser un tableau de taille variable comme un pointeur de pointeur (et inversement).

Dans le cas des tableaux sur le tas : utiliser la librairie libmnitab

`#include "mnitab.h"`

et compiler avec l'option **`-lmnitab`**

$\Rightarrow$ accès aux fonctions du type

`double **`**`double2d`**`(int n, int p)` pour allouer l'espace et

`void` **`double2d_libere`**`(double **mat)` pour libérer l'espace

10 Chaînes de caractères

10.1 Déclaration, affectation des chaînes de caractères

Il n'y a pas de type « chaîne de caractères » en C (contrairement à C++)

$\Rightarrow$ on utilise des tableaux de caractères terminés par **'\0'** (caractère de fin de chaîne)

Chaîne de longueur fixe

```
char st1[4] = "oui" ;
```

un élément de plus pour le **null**

Chaîne de longueur calculée à l'initialisation

```
char *st2 = "oui" ;
```

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)

`sizeof(st2)` produit le résultat `4` (taille de `st2` en mémoire)

10.2 Manipulation des chaînes de caractères

Prototypes des fonctions dans `<string.h>`

Longueur d'une chaîne

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

```
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' + 4 '\0')
printf("%d\n", strlen(ch2));  // => 3 ('n' 'o' 'n')
```

Concaténation de chaînes

```
char *strcat(char *dest,
     const char *src)
```

concatène (ajoute) la chaîne `src` 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`.

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

Noter le format `%s` dans `printf`

Comparaison de chaînes

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

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

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

Recherche d'un caractère dans une chaîne

```
char *strchr(const char *s,
     int c)
```

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

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

Recherche d'une sous-chaîne dans une chaîne

```
char *strstr(
     const char *meule_de_foin,
     const char *aiguille)
```

Renvoie un pointeur vers la première occurrence de la sous-chaîne `aiguille` dans la chaîne `meule_de_foin`, ou `NULL` si `c` n'est pas trouvé.

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

11 Entrées–sorties

11.1 Introduction

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

– lecture des entrées sur le clavier (`stdin`)

– écriture des sorties sur l'écran (`stdout` ou `stderr`)

Pour utiliser des informations archivées (par ex., base de données), ou pour archiver des informations, on a besoin de lire/écrire dans des fichiers.

L'ensemble des prototypes des fonctions ci-dessous est dans `<stdio.h>` (input/output).

11.2 Type de fichiers et accès

fichiers	**formatés**	**binaires**
	saisie et affichage	comme en mémoire
compacité	−	+
rapidité des E/S	−	+
précision conservée	−	+
portabilité	+	−

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

11.3 Ouverture et fermeture d'un fichier

On associe un nom externe de fichier à un nom interne dans le programme (appelé flux).

Ouverture d'un flux

```
FILE *fopen(const char *path,    const char *mode)
```

mode		position
"r"	read	début du fichier
"w"	write	début du fichier
"a"	append	fin du fichier

ajouter

+ si mise à jour (=lecture+écriture)

b si binaire

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

NB : `fopen` rend **`NULL`** si erreur. Toujours tester la valeur retournée par `fopen` pour s'assurer de la bonne ouverture du flux

Fermeture d'un flux (et vidage du tampon)

```
int fclose(FILE *stream)
```

Exemple

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

int main(void) {
FILE *pfich;

pfich=fopen("../rep1/rep2/toto.txt","w");
if (pfich == NULL) {
   printf("Le fichier ../rep1/rep2/toto.txt n'existe pas\n");
   exit(EXIT_FAILURE);
}
// instructions d'écriture
fclose(pfich);
exit(EXIT_SUCCESS);
}
```

11.4 Entrées-sorties non formatées (binaires)

Lecture

```
size_t fread(void *ptr,
    size_t size, size_t nbloc,
    FILE *stream)
```

Lit **nbloc** (le plus souvent 1) de taille `size` dans le flux **stream** (fichier externe) et les écrit à partir de l'adresse `ptr`. **fread** retourne le nombre de blocs effectivement lu $\Rightarrow$ erreur si ce nombre est différent de `nbloc`. Cette erreur peut être la rencontre de la fin du fichier.

Écriture

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

Écrit **nbloc** (le plus souvent 1) de taille `size` situés à l'adresse `ptr` dans le flux **stream** (fichier externe). **fwrite** retourne le nombre de blocs effectivement écrit $\Rightarrow$ erreur si ce nombre est différent de `nbloc`.

11.5 Entrée-sorties formatées

3 fonctions en C pour lire/écrire :

– sur les entrée/sorties standards

– dans les fichiers

– dans les chaînes de caractères

Écriture

stdout	`printf(format [,liste de variables])`
fichier	`fprintf(FILE* stream, format [,liste d'expression])`
chaîne	`sprintf(char* string, format [,liste d'expression])`

NB : en fait `printf` = `fprintf(stdout,...)`

Pour écrire des messages d'erreurs, il vaut mieux utiliser :

`fprintf(stderr,"Erreur...")` que `printf("Erreur...")`

Valeur de retour

Les fonctions de la famille `printf` renvoient un entier :

– égal au nombre de caractères écrits quand tout se passe bien

– négatif, en cas d'erreur (espace disque manquant par exemple)

Lecture

`stdin`	**`scanf(format,liste d'adresses)`**
fichier	**`fscanf(FILE* stream, format,liste de pointeur)`**
chaîne	**`sscanf(char* string, format,liste de pointeur)`**

NB : en fait `scanf` = `fscanf(stdin,...)`

Valeur de retour

La fonction `scanf` renvoie un entier :

– égal au nombre de variables attendues quand tout se passe bien

– inférieur à ce nombre 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.

Si l'on veut vraiment fiabiliser les saisies interactives (`scanf`), il faut **décomposer le processus** en

1. lire ce qui est saisi dans une chaîne de caractères
2. décoder ensuite la chaîne

11.6 Formats d'entrée–sortie

(w =largeur, p= précision)

entiers	
décimal	%w[.p]**d**
octal	%w**o**
hexadécimal	%w**x**
réels	
virgule flottante	%w[.p]**f**
virgule fixe (notation scientifique)	%w[.p]**e**
général (combinaison)	%w[.p]**g**
caractères	
caractères	%w[.p]**c**
chaîne	%w[.p]**s**

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

$\Rightarrow$ risque d'accès mémoire non réservée

11.7 Exemple de lecture de fichier formaté en C

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

```
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;
  }
```

```
  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) ;
}
```

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.
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.

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

Structures statiques : champs de taille fixe

Structures dynamiques : comportant des champs de taille variable

Les structures sont la base des « objets » dans les langages orienté objet (objet $=$ structure $+$ méthodes), dont C++.

12.2 Définition, déclaration et initialisation des structures

Définition d'un type structure `point`

```
struct point {
        int no ;  //1er champ
        float x ; //2e champ
        float y ; //3e champ
};
```

Noter le `;` obligatoire après l'accolade `}`.

Déclaration de variables de type structure `point`

Une structure se comporte comme un nouveau type du langage.

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

Initialisation d'une structure (constructeur)

L'initialisation globale peut se faire à la déclaration (comme pour un tableau)

```
struct point fin={9,5.,2.};
```

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

Tous les types de base du langage (`char`, `int`, `double`...)

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 ($\Rightarrow$ Pas de tableau automatique)

Des structures (sauf du type en train d'être défini)

Où déclarer une structure ?

La déclaration de la structure doit être visible dans toutes les fonctions utilisant le type structure déclaré. On doit donc déclarer 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

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 ;
debut.x = 0. ;
debut.y = 0. ;
```

$\Rightarrow$ L'accès aux différents champs par leur nom est plus lourd que l'accès aux éléments d'un tableau (pas d'indice $\Rightarrow$ pas de boucle), mais il permet de rendre parfois les programmes plus clair (nommage)

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 (<, >...)

Entrées/sorties

obligatoirement champ par champ :

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

12.4 Représentation en mémoire des structures

Les champs des structures sont stockés dans l'ordre de leur déclaration et à proximité les uns des autres en mémoire. Mais (contrairement aux tableaux) il peut exister des « octets de remplissage » entre les différents champs, afin de repecter des contraintes d'alignement.

$\Rightarrow$ la taille (au sens de `sizeof`) d'une structure $>=$ la somme des tailles de ses différents éléments.

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

fournit le résultat 8 $\Rightarrow$ 2 octets de remplissage (sur une machine 32 bits) entre `no` et `x`

Pointeur sur une structure

On peut définir un pointeur sur une structure :

```
struct mini_point mpo, *pmpo;
pmpo=&mpo;
```

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

`pmpo` vaut `&mpo.no`

12.5 Exemples de structures (plus ou moins) complexes

Tableaux de structures

```
struct point nuage[9] ;
courbe[0].x = 2. ;
```

abscisse du premier point du nuage

Structures contenant un tableau (taille fixe)

```
struct courbe {
   double x[10];
   double y[10];
};
...
struct courbe c;
c.x[0]=0.5;
c.y[0]=sqrt(c.x[0]);
```

Structures contenant un tableau (taille variable)

Les tableaux automatiques (sur la pile) sont impossibles à utiliser au sein des structures.

Uniquement taille variable sur le tas ⇒ pointeurs et `malloc` ou `calloc`

```
struct courbe {
   int n;
   double *x;
   double *y;
};
...
struct courbe c;
scanf("%d", &c.n); // nb 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]);
```

Attention, dans ce cas, aux copies superficielles (uniquement le pointeur)

```
struct courbe c1, c2;
...
c2 = c1;
free(c1.x); // le pointeur c2.x ne pointe plus vers une zone
```

Solution : Il faut donc allouer un espace mémoire pour `c2.x` et copier tous les éléments du tableaux

```
c2 = c1 ;
c2.x=(double *)calloc(c2.n, sizeof(double)); //allocation
for (int i=0; i < c2.n; i++) {
   c2.x[i] = c1.x[i];
}
free(c1.x); // le pointeur c2.x continue de pointer vers une
```

Structures contenant une structure

```
struct lieu {
   char nom[80];
   double longitude;
   double latitude;
};
struct observation {
   struct lieu ville;
   double temperature;
};
...
struct observation obs;
obs.ville.longitude=0.; // association de "."
obs.ville.latitude=45.;
obs.temperature=273.15;
sprintf(obs.ville.nom,"%s","Bordeaux");
```

Liste chaînées

Ce sont des structures contenant un pointeur vers une structure du même type

```
struct point{
    float x;
    float y;
    struct point *next;
};
```

Elles permettent de mettre en œuvre des « structures » mathématiques de type graphe, arbre, etc.

12.6 Structure et fonction, opérateur flêche

Les structures se comportent comme n'importe quel type du langage.

Transmission par 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("%g %g\n", p.x, p.y);
}
...
struct point po;
...
affiche(po);
```

Transmission par adresse (ou référence)

À utiliser quand on veut modifier l'argument dans la fonction

```
void init_point(struct point *p) {
    (*p).x=0.;
    (*p).y=0.;
}
...
struct point po;
...
init_point(&po);
```

Opérateur flêche

On évite dans ce cas la notation lourde `(*p_struct).champ` en utilisant 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->x=0.;
    p->y=0.;
}
```

Valeur de retour

une fonction peut retourner une structure (elle est alors du type `struct nom_struct`).

```
struct point cree_point(double x, double y) {
   struct point p;
   p.x=x;
   p.y=y;
   return p;
}
...
struct point po;
double x0, y0;
...
po=cree_point(x0, y0);
```

12.7 Exemple final

```
// Fichier point.h
struct point { // définition du type point
  int no;  /* numéro */
  float x; /* abscisse */
  float y; /* ordonnee */
};

// fichier point.c
#include <stdio.h>
#include <stdlib.h>
#include "point.h"
```

```
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;
}
```

```c
int main(void){
  struct point a ={5, 1. ,-2.}; // définition 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);
}
```

```
a = 5 1 -2
psym(a) = -5 -2 1
sym. de a = -5 -2 1
```

13 Éléments de compilation séparée

13.1 Introduction

Il est d'usage de séparer les programmes « long » en plusieurs fichiers :

– lors de la conception, il est plus facile de compiler séparément des entités de programme courtes que des centaines de lignes de code (ou plus)
– lors de la conception, la séparation en plusieurs fichiers permet également de structurer un programme $\Rightarrow$ fichier = unité cohérente
– le découpage en unités cohérentes permet enfin une ré-utilisation plus facile du code

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...`) partout où cela est nécessaire

Dans les 2 cas, la solution réside dans l'utilisation de fichiers d'entête.

13.2 Fichiers d'entête

Ce sont les fichiers introduits par la directive #**include** du préprocesseur.

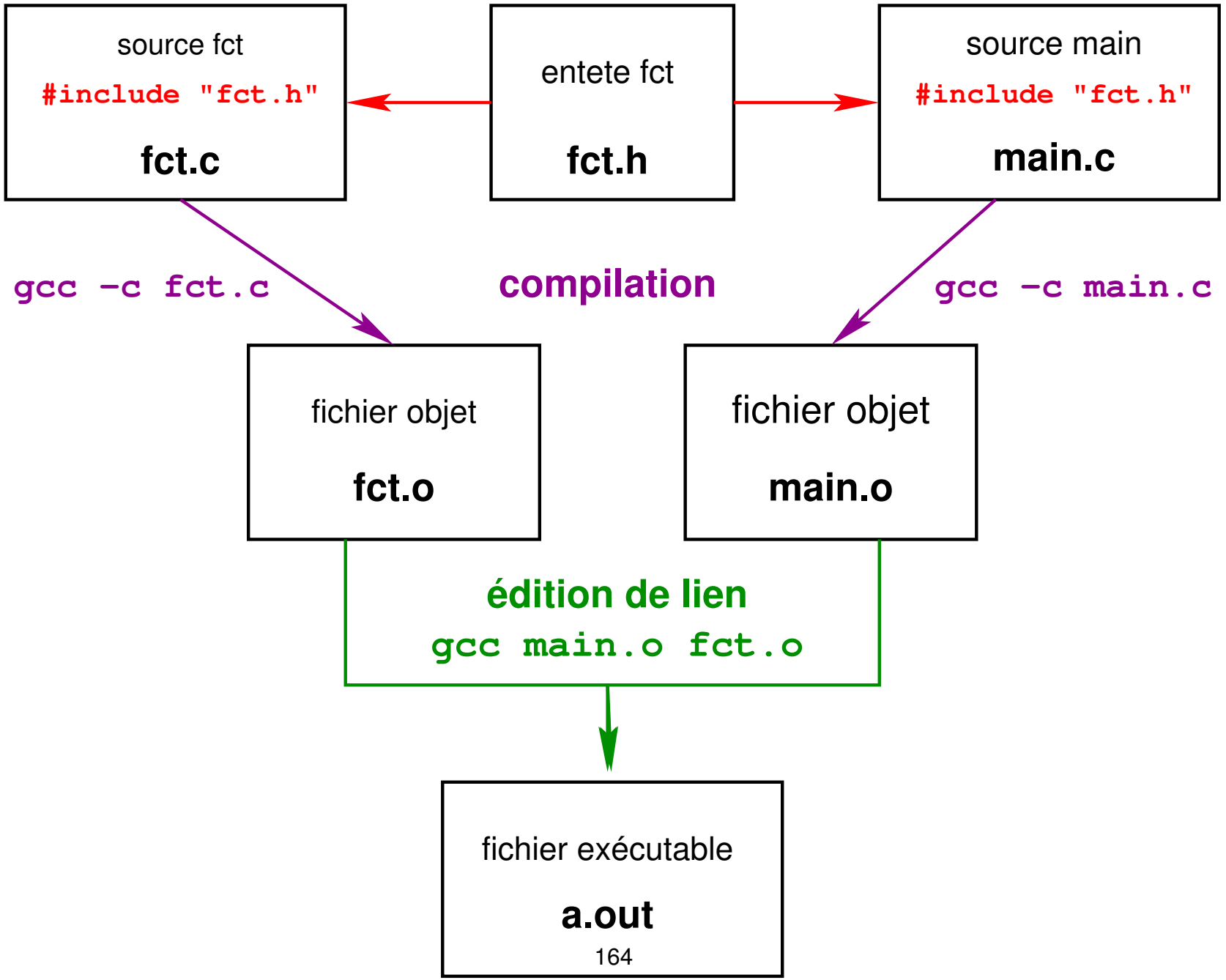
Ces fichiers peuvent contenir des déclarations de fonctions (prototypes) ou de nouveaux types.

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/définition

Pour les fonctions comme pour les types, il faut se protéger contre les inclusions multiples : les redéclarations sont interdites.


source fct
#include "fct.h"
fct.c
entete fct
fct.h
source main
#include "fct.h"
main.c
gcc -c fct.c
compilation
gcc -c main.c
fichier objet
fct.o
fichier objet
main.o
édition de lien
gcc main.o fct.o
fichier exécutable
a.out

13.3 Structure d'un fichier d'entête

Soit un fichier `fct.c` contenant la définition de plusieurs fonctions. Le fichier `fct.h` peut s'écrire :

```
#ifndef FCT
#define FCT
void calcul1(int x, double *res);
double calcul2(double x, double y);
#endif
```

13.4 Compléments sur les fonctions

La Récursivité des fonctions (=fonction qui s'appelle elle-même) est automatique en C.

13.4.1 Exemple de fonction récursive : factorielle

Factorielle récursive en 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 ;
}
return factorielle;
}
```

version idiomatique : `return n>0 ? n*fact(n-1) : 1;`