INITIATION AU LANGAGE C

PRÉSENTÉ PAR N. FACI

INTRODUCTION (2)

Enoncé du problème

Algorithme

Programme en Langage de programmation évolué

Compilation + exécution

Solution du problème

PARTIE 1 : BASES DE LA PROGRAMMATION EN C

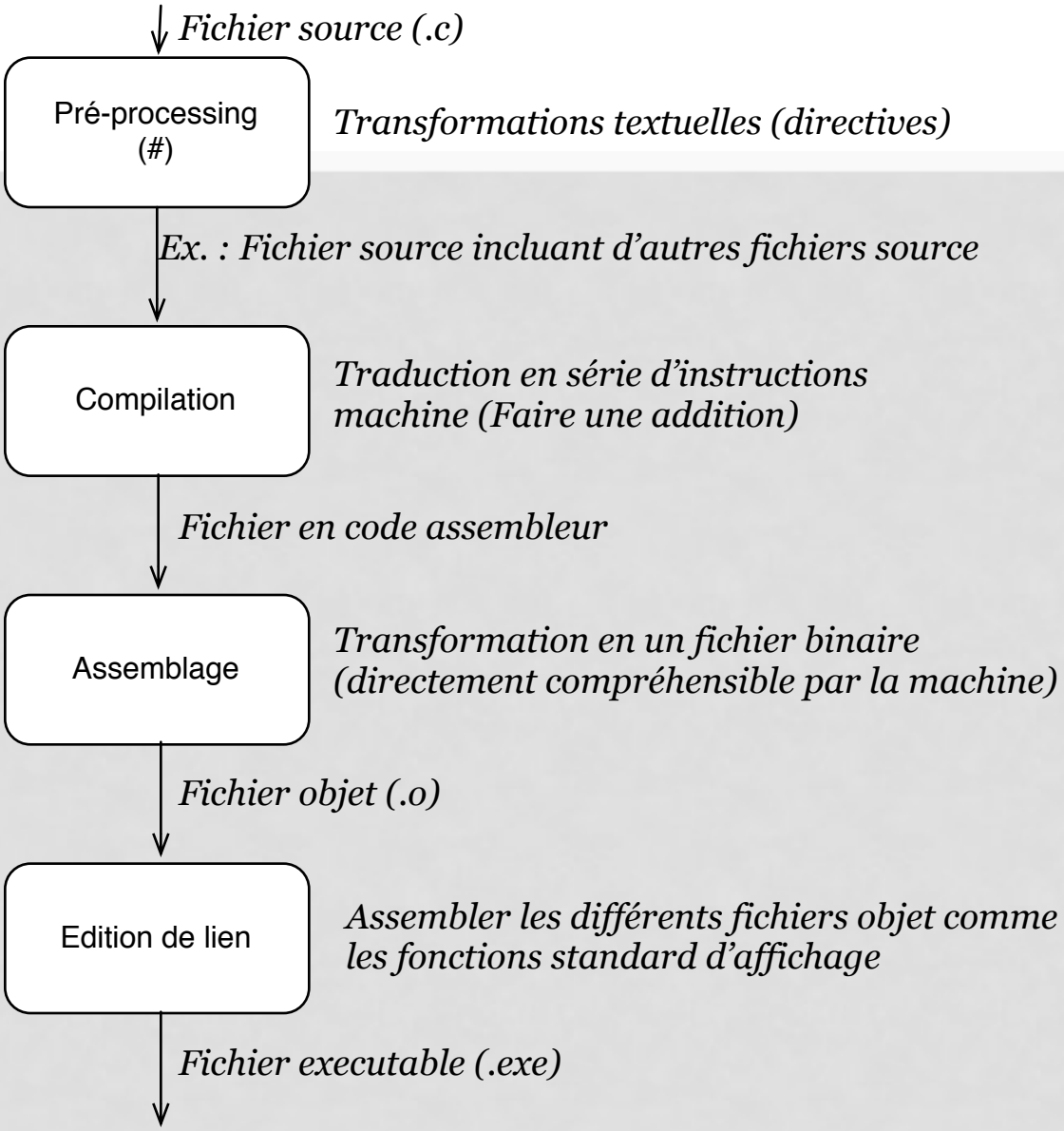
PREMIER PROGRAMME EN C

```
#include<stdio.h>
int main()
{
   printf("Ceci est mon premier programme en C\n");
   return 0;
}
```

- Un programme C comporte une fonction principale appelée **main()** renfermant les instructions qui doivent être exécutées

- **printf(…)** produit une émission de caractères vers la sortie standard (l'écran par défaut)
  - Entré: chaines de caractères entre une paire de double quotte
- **stdio.h** définit l'usage de la fonction printf
- **return 0** valeur renvoyée au système d'exploitation en fin de programme
  - Valeur 0 correspond à une sortie « sans erreur » du programme

4 Phases de compilation



ERREUR ET AVERTISSEMENT À LA COMPILATION

Le compilateur affiche deux types de messages

- **Errors** : la compilation a échoué en un point du programme source, parce que la syntaxe du langage c n'a pas été respecté.
  - un message en anglais concernant la nature de l'erreur détectée avec le numéro de la ligne ou se trouve l'erreur.
- **Warnings**: est un avertissement. Le compilateur a réalisé le travail mais il signale qu'il a détecté un problème potentiel quand on exécute le programme.

COMPOSANTS ÉLÉMENTAIRES DU C

Identificateurs

Mots clé

Constantes

Commentaires

Chaines de caractères

Opérateurs

Signes de ponctuation

IDENTIFICATEURS

- Un identificateur est une suite de caractères parmi:
  - Les lettres (minuscules, majuscules, non accentuées)
  - Les chiffres
  - Le « blanc souligné » (_)
- Le premier caractère d'un identificateur ne peut être un chiffre.
- Le rôle d'un identificateur est de donner un nom à une entité du programme. Plus précisément, un identificateur peut entre autres désigner:
  - Un nom de variable ou de constante ou de fonction

MOTS CLEFS

- Un certain nombre de mots, appelés mots clefs, sont réservés pour le langage lui-même et **ne peuvent être utilisés comme identificateurs**.

- Sont rangés en plusieurs catégories:

  - Spécificateurs de type:

  char double enum float int long short signed unsigned void

  - Instructions de contrôle:

  If else while

  - Divers: return, **sizeof**

STRUCTURE D'UN PROGRAMME EN C

- Un programme en C se présente de la façon suivante:

[*directives au pré-processeur*]
[*déclarations de variables globales*]
[*fonctions secondaires*]

void main()
{
déclarations de variables locales
instructions
}

STRUCTURE D'UN PROGRAMME EN C

- **Instruction de déclaration:** spécificateur de type et d'une liste d'identificateurs séparés par une virgule, suivie d'un point virgule.

  Ex: Int a;

NB: En C, toute variable doit faire l'objet d'une déclaration avant d'être utilisée.

- **Instruction d'affection**: elle est composée d'un identificateur de variable suivi de l'opérateur « =» et d'une expression arithmétique et/ou logique
  - x=2.38 e 4;

- **Instructions composées**: blocs conditionnels, itératifs

STRUCTURE D'UN PROGRAMME EN C

- Un programme en C se présente de la façon suivante:

[*directives au pré-processeur*]
[*déclarations de variables globales*]
[*fonctions secondaires*]

```
int main()
{
déclarations de variables locales
Instructions
return 0;
}
```

STRUCTURE D'UN PROGRAMME EN C

- Les fonctions secondaires peuvent être placées avant ou après le main.
- Elles se décrivent de la manière suivante:

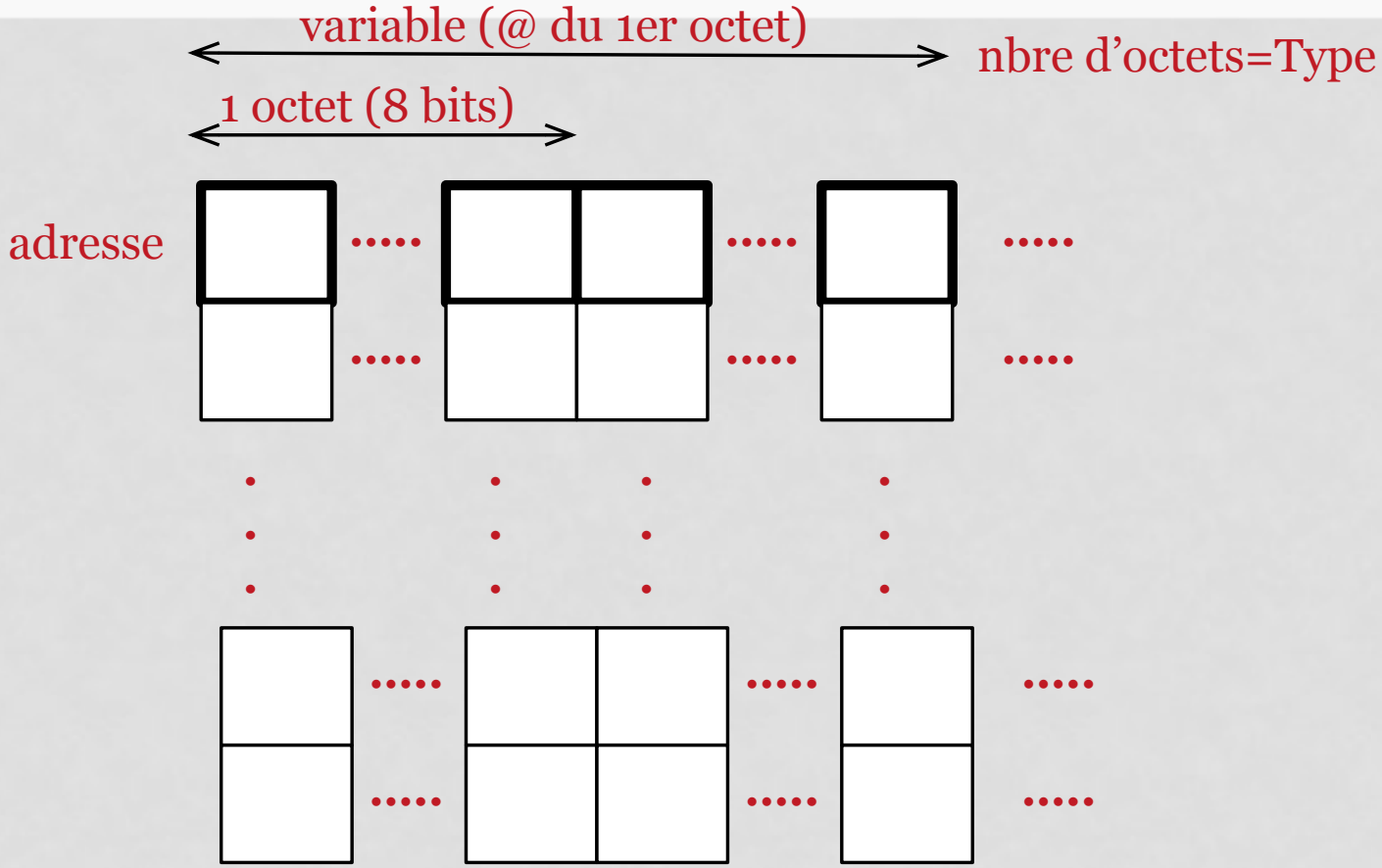
type nom_fonction(*arguments*)
{
déclarations de variables locales
instructions
}

STRUCTURE D'UN PROGRAMME EN C

- Exemple:

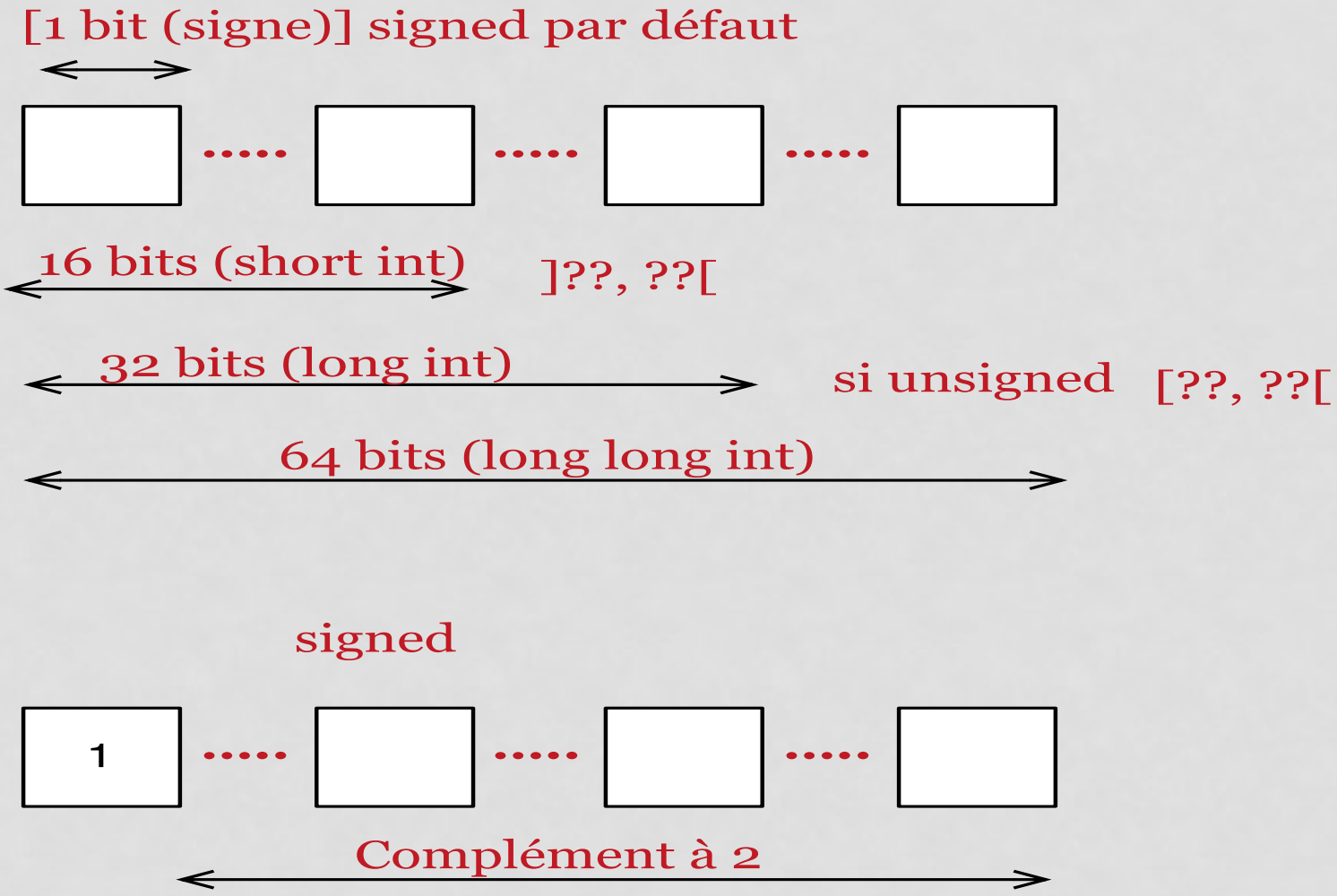
```
/* pour qu'on puisse utiliser les fonctions de la bibliothèque standard stdio.h*/
#include <stdio.h>

int main() { /*: début du programme */

printf ("Hello, World!\n"); /* appel de la fonction printf de la biblio stdio.h*/

return 0; } /* valeur renvoyée au système d'exploitation en fin de programme*/
/* valeur 0 correspond à une sortie « sans erreur » du programme*/
```

VARIABLES



Mémoire centrale

LES TYPES ENTIERS



LES TYPES FLOTTANTS

Partie Entière **Partie Exposant**

123.456E-78

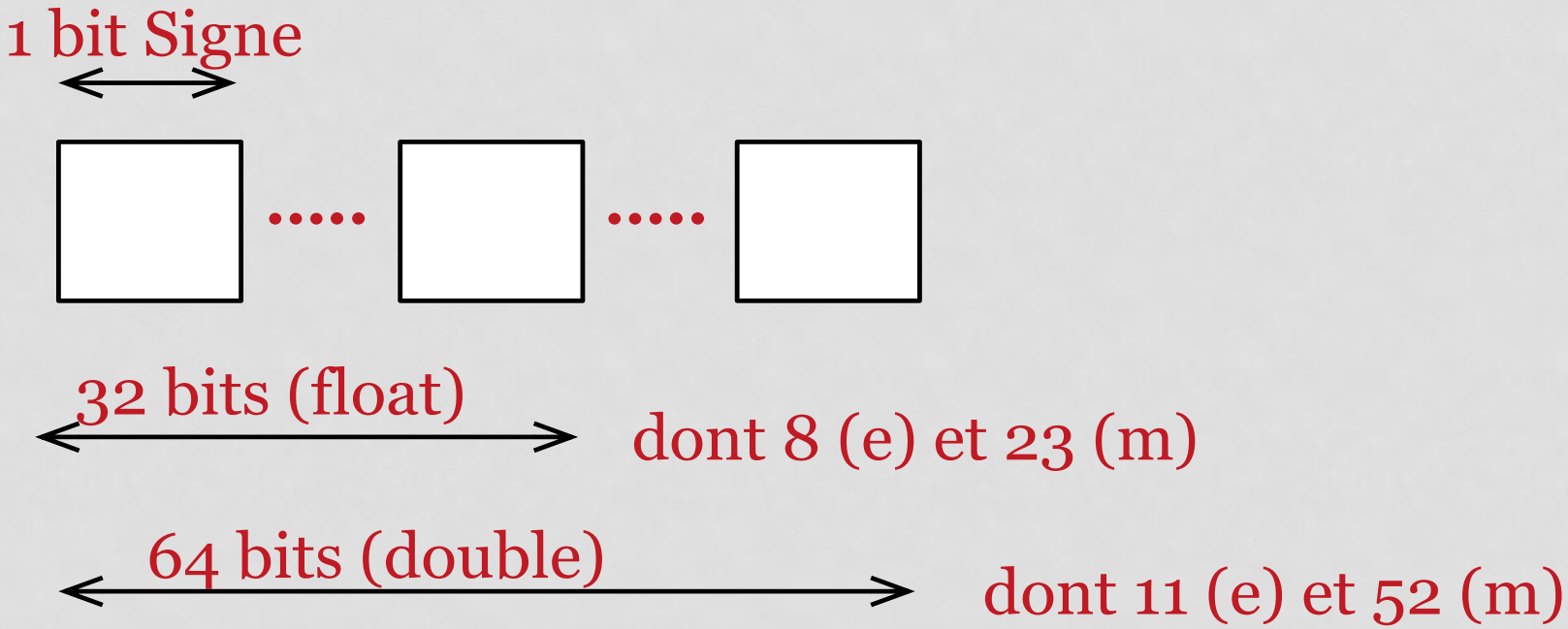
Partie Fractionnaire

mantisse (m) exposant (e)

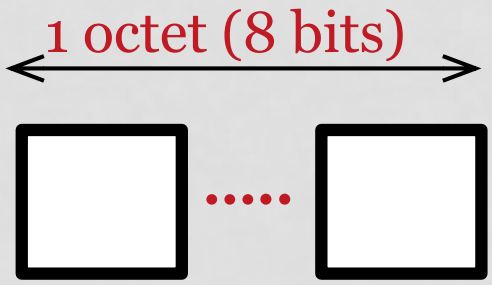
123.456E-78

1234.56E-77

Codage


1 bit Signe
32 bits (float)
dont 8 (e) et 23 (m)
64 bits (double)
dont 11 (e) et 52 (m)

LE TYPE CARACTÈRE



1 caractère, code ASCII (0, 255)

Déclaration: char c;
Valeurs: ‘A’, ‘a’, …, ‘Z’, ‘z’

CONSTANTES

- Les constantes sont des valeurs fixes et ne changent pas au cours de l'exécution de programme

- Constantes entières:
  - Décimale: 0 et 2435678
  - Octale: 0 et 255 -> 00 et 0377
  - Héxa-décimale: Oxe (pour 14)
  - Non signée et Longue: 123U et 123456UL

CONSTANTES

- Constantes réelles:

12.34 (double), 12.3e-4 (double),12.34F (float), 12.34L (long double)

- Constantes caractères et chaîne de caractères :

'a'…'z', 'A'…'Z', « ceci est une chaine de caractères », caractères non imprimables (\n, \t, \r, etc.)

- Les constantes numériques peuvent être précédées par le signe –

- Les variables de type « const » ne peuvent être modifiées

FONCTIONS CLASSIQUES D'ENTRÉE/ SORTIE (1)

- Fonctions de la librairie standard **stdio.h** utilisées avec les unités classiques d'entrées-sorties, qui sont respectivement le clavier et l'écran.

- Deux fonctions principales:

  - Fonction d'impression formatée, ce qui signifie que les données sont converties selon le format particulier choisi: **printf**
  - Fonction de saisie: **scanf**

FONCTION PRINTF (1)

- Syntaxe de **printf** est:

  printf("*chaîne de contrôle ",expression-1, ..., expression-n);*

- chaîne de contrôle: le texte à afficher et les spécifications de format correspondant à chaque expression de la liste.

- Les spécifications de format ont pour but d'annoncer le format des données à visualiser. (voir tableau), introduites par le caractère **%**, suivi d'un caractère désignant le format d'impression.

FONCTION PRINTF (2)

format	conversion en	écriture
%d	int	décimale signée
%ld	long int	décimale signée
%u	unsigned int	décimale non signée
%lu	unsigned long int	décimale non signée
%o	unsigned int	octale non signée
%lo	unsigned long int	octale non signée
%x	unsigned int	hexadécimale non signée
%lx	unsigned long int	hexadécimale non signée
%f	double	décimale virgule fixe
%lf	long double	décimale virgule fixe
%e	double	décimale notation exponentielle
%le	long double	décimale notation exponentielle
%g	double	décimale, représentation la plus courte parmi %f et %e
%lg	long double	décimale, représentation la plus courte parmi %lf et %le
%c	unsigned char	caractère
%s	char*	chaîne de caractères

FONCTION PRINTF(3)

- On peut éventuellement préciser certains paramètres:
  - *Largeur maximale d'impression: %10d*
  - *Précision:*
    - *%.12f : un flottant sera imprimer avec 12 chiffres après la virgule*
    - *%10.2f: réserver 12 caractères pour imprimer avec '.' compris dont 2 sont destinés aux chiffres après la virgule*
    - *%30.4s: 30 caractères pour imprimer la chaîne mais seuls les 4 premiers caractères de la chaines seront imprimés suivis de 26 espaces blanc.*

FONCTION PRINTF(4)

```
#include <stdio.h>
int main()
{
    int i = 23674;
    double x = 1e-8 + 1000;
    char c = ’A’;
    printf("impression de i: \n");
    printf("%d \t %u \t %o \t %x",i,i,i,i);
    printf("%f \t %e \t %g",x,x,x);
    printf("\n%.2f \t %.2e",x,x);
    printf("\n%.20f \t %.20e",x,x);
    printf("\nimpression de c: \n");
    printf("%c \t %d",c,c);
    return 0;
}
```

FONCTION SCANF(1)

- **scanf** permet de saisir des données au clavier et de les stocker aux adresses spécifiées par les arguments de la fonction:

  scanf("*chaîne de contrôle*",*argument1,...,argumentN)*

- *chaîne de contrôle : format dans lequel les données lues sont converties. Elle ne* contient pas d'autres caractères (notamment pas de \n).

FONCTION SCANF(2)

- Comme pour printf, les conversions de format sont spécifiées par un caractère précédé du signe %.
- Les formats valides pour scanf diffèrent légèrement de ceux de la fonction printf.
- Les données à entrer au clavier doivent être séparées par des blancs ou des <RETURN> sauf s'il s'agit de caractères.
- On peut toutefois fixer le nombre de caractères de la donnée à lire.
- Par exemple %3s pour une chaîne de 3 caractères, %10d pour un entier qui s´´etend sur 10 chiffres signe inclus.

FONCTION SCANF(3)

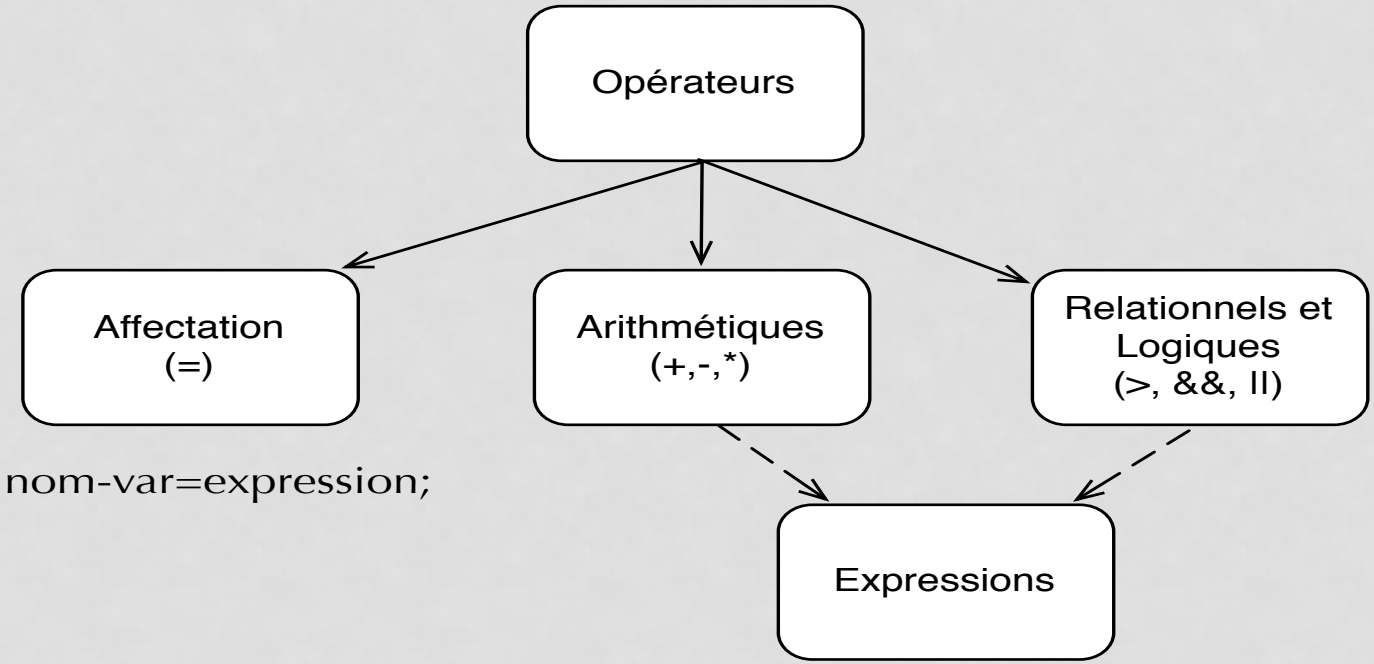
format	type d'objet pointé	représentation de la donnée saisie
%d	int	décimale signée
%hd	short int	décimale signée
%ld	long int	décimale signée
%u	unsigned int	décimale non signée
%hu	unsigned short int	décimale non signée
%lu	unsigned long int	décimale non signée
%o	int	octale
%ho	short int	octale
%lo	long int	octale
%x	int	hexadécimale
%hx	short int	hexadécimale
%lx	long int	hexadécimale
%f	float	flottante virgule fixe
%lf	double	flottante virgule fixe
%Lf	long double	flottante virgule fixe
%e	float	flottante notation exponentielle
%le	double	flottante notation exponentielle
%Le	long double	flottante notation exponentielle
%g	float	flottante virgule fixe ou notation exponentielle
%lg	double	flottante virgule fixe ou notation exponentielle
%Lg	long double	flottante virgule fixe ou notation exponentielle
%c	char	caractère
%s	char*	chaîne de caractères

FONCTION SCANF(4)

```
#include <stdio.h>
int main()
{
   int i;
   printf("entrez un entier sous forme hexadecimale i = ");
   scanf("%x",&i);
   printf("i = %d\n",i);
   return 0;
}
```

EXPRESSIONS

OPÉRATEURS


Opérateurs
Affectation
(=)
Arithmétiques
(+,-,*)
Relationnels et
Logiques
(>, &&, ||)
nom-var=expression;
Expressions
nom-var+nom-var
nom-var+const
nom-var+nom-const
nom-var&&nom-var
nom-var>nom-var
nom-var>const

OPÉRATEURS

```
#include <stdio.h>
int main(){
   int i,j=2;
   float x=2.5;
   i=j+x;
   printf(« \n %f \n »,x);
   printf(« \n %d\n »,i);
   return 0;
}
```

OPÉRATEURS ARITHMÉTIQUES

- Opérateurs arithmétiques sont l'opérateur unaire (- changement de signe) ainsi que les opérateurs binaires:
  - **-** soustraction: a-b,
  - **+** addition: a+b,
  - ***** multiplication: a*b,
  - **/** division: a/b,
  - **%** reste de la division (modulo): a%b.

OPÉRATEURS ARITHMÉTIQUES

- Quelques spécificités du C
  - float x; x=3/2; affecte à x la valeur 1

  Par contre, float x; x=3/2.; affecte à x la valeur 1.5
  - % ne s'applique qu'à des opérandes de type entier
  - Pas d'opérateur élévation à la puissance. Il faut utiliser **pow(x,y)** de la librairie **math.h** pour calculer $\mathbf{x^y}$.

OPÉRATEURS RELATIONNELS

- Opérateurs relationnels :
  - > strictement supérieur
  - < strictement inférieur
  - >= supérieur ou égal
  - <=inférieur ou égal
  - == égal
  - != différent

OPÉRATEURS RELATIONNELS

- Syntaxe : <expression1> op <expression2>
- Les deux expressions sont évaluées puis comparées.
- La valeur rendue est de type int (**il n'y a pas de type booléen en C**); elle vaut 1 si la condition est vraie, et 0 sinon

OPÉRATEURS RELATIONNELS

```
#include <stdio.h>
int main()
   {
        int a = 0;
        int b = 1;
        if (a = b)
                printf("\n a=%d et b=%d sont egaux  \n », a,b);
        else
                printf("\n a=%d et b=%d sont differents \n », a, b);
        return 0;
}
```

imprime à l'écran a et b sont egaux ! Remplacer = par ==

OPÉRATEURS LOGIQUES BOOLÉENS

- On retrouve :
  - **&&** et logique,
  - **||** ou logique,
  - **!=** différent,
  - **!** Négation
- Syntaxe : <expression1> op1 <expression2> op2 …. <expressionN>
- L'évaluation se fait de gauche à droite et s'arrête dès que le résultat final est déterminé.

Par ex. (i>=0) && (i<=9) && (p[i]==0): la dernière clause ne sera pas évaluée si i n'est pas compris entre 0 et 9.

OPÉRATEURS LOGIQUES BIT À BIT

- On retrouve :
  - **&** et,
  - **|** ou inclusif,
  - **^** ou exclusif,
  - **~** complément à 1
  - **<<** décalage à gauche (multiplication par puissance de 2), **>>** décalage à droite (division par puissance de 2)

OPÉRATEURS LOGIQUES BIT À BIT

Expression	Valeur binaire	Valeur décimale
a	01001101	77
b	00010111	23
a & b	00000101	5
a | b	01011111	95
a ^ b	01011010	90
~ a	10110010	178
b << 2	01011100	92
b << 5	11100000	112
b >> 1	00001011	11

Multiplication par 4,
ce qui dépasse disparaît

OPÉRATEURS D'AFFECTATION COMPOSÉE

On retrouve :

- **+= , -=, *= , /= , %= , &= , ^= , |= , <<= , >>=**

- Pour tout opérateur op, l'expression:

expression1 op= expression2 ⇔ expression1=expression1 op expression2

Exemple: x+=1; x%=2;

OPÉRATEURS D'INCRÉMENTATION ET DE DÉCRÉMENTATION

- On retrouve :
  - **++** incrémentation
  - **--** décrémentation
- Ils s'utilisent aussi bien en préfixe qu'en suffixe

Exemples: i++, ++i, i--, --i

OPÉRATEURS D'INCRÉMENTATION ET DE DÉCRÉMENTATION

```
#include<stdio.h>
int main() {
 /* Evaluation de l'expression de gauche à droite*/
int a=3, b, c;
b=++a;
printf(" \n la valeur de b est %d \n ",b);
c=b++;/* b vaut ?? Et c vaut ?? */
printf("\n la valeur de a est %d \n ",a);
printf(" \n la valeur de b est %d \n ",b);
printf("\n la valeur de c est %d \n ",c);
b++;
printf(" \n la valeur de b est %d \n ",b);    r
eturn 0;
}
```

OPÉRATEUR DE CONVERSION DE TYPE

- Appelé ***cast***, modifie explicitement le type d'une variable.

(type) *nom-vart*

```
int main()
{
   int i = 3, j = 2;
   float x= (float) i/j;
   /*printf("%f \n",(float)i/j);*/
   printf(«  %d %f \n",i, x);
   return 0;
   }
```

retourne la valeur 1.5

AUTRES OPÉRATEURS

- L'opérateur d'adresse **&** appliqué à une variable retourne l'adresse-mémoire de cette variable : **&***variable*

- L'opérateur **sizeof** retourne la longueur en octets du type spécifié comme paramètre: **sizeof(type)**

RÈGLES DE PRIORITÉ ENTRE OPÉRATEURS

opérateurs	
`()` `[]` `->` `.`	→
`!` `~` `++` `--` -(unaire) (*type*) `*`(indirection) `&`(adresse) `sizeof`	←
`*` `/` `%`	→
`+` -(binaire)	→
`<<` `>>`	→
`<` `<=` `>` `>=`	→
`==` `!=`	→
`&`(et bit-à-bit)	→
`^`	→
`|`	→
`&&`	→
`||`	→
`? :`	←
`=` `+=` `-=` `*=` `/=` `%=` `&=` `^=` `|=` `<<=` `>>=`	←
`,`	→

INSTRUCTIONS DE CONTRÔLE

INSTRUCTIONS DE CONTRÔLE

- On appelle « **Instruction de contrôle** » toute instruction qui permet de contrôler le fonctionnement d'un programme.
- On distingue les instructions de branchement et les boucles.
- Une instruction de branchement permet de déterminer quelles instructions seront exécutées et dans quel ordre
- Une boucle permet de répéter une série d'instructions tant qu'une certaine condition n'est pas vérifiée.

INSTRUCTIONS DE BRANCHEMENT CONDITIONNEL (1)

- Branchement conditionnel **if --- else**

Forme la plus simple	Forme la plus générale

if (expression)
  instruction
	if (expression1)
	  instruction1
	else if (expression2)
	  instruction2
	…
	else if (expressionN)
	  instructionN
	else
	  instructionX

Chaque instruction peut être un bloc d'instruction

INSTRUCTIONS DE BRANCHEMENT CONDITIONNEL (2)

- Branchement multiple **switch**

Si la valeur de *expression* est égale à l'une des *constantes, la liste d'instructions* correspondant est exécutée. Sinon la *liste d'instructions X* correspondant à *default* est exécutée.
L'instruction *default* est facultative.

Forme la plus générale

```
switch (expression)
    {
     case constante1:
          liste d'instructions 1
          break;
     case constante2:
          liste d'instructions 2
          break;
…
     case constanteN:
          liste d'instructions N
          break;
      default:
          liste d'instructions X
          break;
     }
```

INSTRUCTIONS DE BRANCHEMENT CONDITIONNEL (2)

- Exemple

Ecrire un programme qui saisit soit deux valeurs entières soit deux valeurs réelles, calcule leur somme et l'affiche.

BOUCLES (1)

- Boucle **while**

	Forme
	while (*expression*) *instruction*

Tant que *expression* est vérifiée (i.e., non nulle), *instruction* est exécutée. Si *expression* est nulle au départ,

ne sera jamais exécutée. *instruction* peut évidemment être une instruction composée.

```
i=1;
while (i < 10)
    {
      printf(« \n i = %d »,i);
       i++;
     }
```

BOUCLES (2)

- Boucle **do---while**

	Forme
	do *instruction* while (*expression*);

Il se peut que l'on souhaite effectuer le test de continuation qu'après avoir l'instruction. Dans ce cas, on utilise la do---while. Ici, *instruction* sera exécutée tant que *expression* est non nulle.

```
int a;
do
   {
      printf(« \n Entrez un entier entre 1 et 10: »);
      scanf(« %d »,&a);
    }
while ((a<=0) || (a>10));
```

BOUCLES (3)

- Boucle **for**

Syntaxe	Version Equivalente

for (expr1 ; expr2; expr3)
instruction
	Expr1;
	while *(expr2)*
	{
	instruction
	expr3;
	}

```
int i;
for (i=0;i<10;i++)
    printf(« \n i =
```

```
int n,i,fact;
scanf(« %d »,&n) ;
for (i=1,fact=1;i<=n;i++)
   fact*=i;
printf(«  %d ! = %d \n»,n,fact);
```

```
int n,i,fact;
 for (i=1,fact=1;i<=n;fact*=i,i++);
printf(«  %d ! = %d \n»,n,fact);
```

BRANCHEMENT NON CONDITIONNEL *BREAK*

```
#include <stdio.h>
#include<math.h>
int main()
{
     int i,j;
     for (i=3;i<=100;i=i+1)
     {
        for (j=2;j<=i;j=j+1)
       {
           if (i%j==0)
                 {
                     return 1;
                     break;
                 }
           if (j>sqrt(i))
                 {
                     printf(«  %d \n»,i);
                     return 1;
                     break;
                  }
              }
         }
        return 0;
}
```

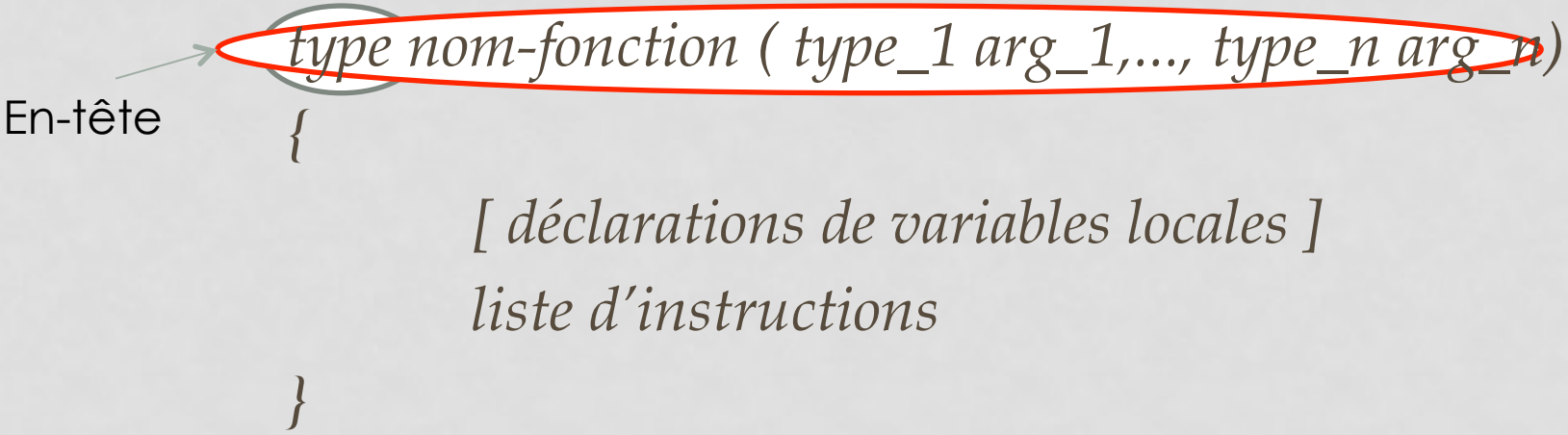
NOTIONS DE BASE SUR LES FONCTIONS (1)

- En C, on peut découper un programme en plusieurs parties, appelées fonctions.
- Une seule de ces fonctions existe **obligatoirement**; c'est la fonction principale appelée «main».
- Les autres sont appelées «fonctions secondaires», qui à leur tour peuvent faire appel à d'autres fonctions.

FONCTIONS

DÉFINITION D'UNE FONCTION(1)

- Est la donnée du texte de son algorithme, qu'on appelle corps de la fonction.



DÉFINITION D'UNE FONCTION (2)

- En C, il n'y a pas la notion de « procédure »
- Une fonction qui ne retourne pas de valeur est une fonction dont le type est spécifié par «void»
- Les arguments de la fonction s'appellent « paramètres formels », contrairement aux « paramètres effectifs » qui sont les paramètres avec lesquels est appelée la fonction.

DÉFINITION D'UNE FONCTION (3)

- Les paramètres formels peuvent être de n'importe quel type
- Leurs identificateurs n'ont d'importance qu'à l'intérieur de la fonction
- Si la fonction n'a pas de paramètres formels, on peut remplacer la liste des paramètres par le mot-clé «void»

DÉFINITION D'UNE FONCTION (4)

- Le corps de la fonction débute éventuellement par la déclaration de variables, qui sont locales à la fonction et ne peuvent par conséquent pas être utilisées par d'autres fonctions.

- Il se termine par l'instruction de retour à la fonction appelante «return» : return(*expression*); où « expression » est la valeur que retourne la fonction.

- Si la fonction ne retourne pas de valeur, on peut mettre return;

EXEMPLES DE FONCTIONS

```
int produit (int a, int b)
{
   return(a*b);
}
int puissance (int a, int n)
{
   if (n == 0)
         return(1);
   return(a * puissance(a, n-1));
}
```

APPEL DE FONCTIONS

- Se fait par l'expression:

  nom-fonction(para_1,para_2,...,para_n)

- L'ordre et le type des paramètres effectifs de la fonction doivent concorder avec ceux dans l'en-tête de la fonction.

- Par exemple:
  - produit(2,4); mais pas produit(2.,4)
  - void main ()
    {
        int a,b;
        produit(b,a); (Qu'en pensez-vous?)
    }

DÉCLARATION DE FONCTIONS (1)

- C n'autorise pas l'imbrication de fonctions.
- La définition d'une fonction secondaire doit donc être placée avant ou après la fonction « main ».
- Toutefois, il est indispensable que le compilateur connaisse la fonction au moment où elle est appelée.
- Si une fonction est définie après son premier appel, elle doit être impérativement déclarée avant.

DÉCLARATION DE FONCTIONS (2)

- Une fonction secondaire est déclarée par son prototype, qui donne le type de la fonction et celui de ses paramètres:

type nom-fonction(type_1,...,type_n);

```
int puissance (int, int );
void main()
{
    int a = 2, b = 5;
    printf("%d\n", puissance(a,b));
}
int puissance (int a, int n)
{
    if (n == 0)
    return(1);
    return(a * puissance(a, n-1));
}
```

DÉCLARATION DE FONCTIONS (3)

- Les fichiers d'extension .h de la librairie standard (fichiers header) contiennent notamment les prototypes des fonctions.
- Par exemple, dans math.h, on retrouve le prototype de la fonction pow: double pow(double,double);
- #include <math.h> permet au compilateur d'inclure la déclaration de la fonction pow dans le fichier source
- Ainsi, si pow est appelé avec des paramètres de type int, ces paramètres seront convertis en double lors de la compilation.

DURÉE DE VIE DES VARIABLES (1)

- Deux types de variable : permanent et temporaire
- **Variables permanentes (ou statiques)**:
  - Elles occupent un emplacement en mémoire qui reste le même durant toute l'exécution du programme.
  - Elles sont initialisées par défaut à zéro.
  - Elles sont caractérisées par le mot-clef static.
  - Zone mémoire: **Segment de données**

DURÉE DE VIE DES VARIABLES (1)

- **Variables temporaires**:
  - Elles se voient allouer un emplacement en mémoire de façon dynamique lors de l'exécution du programme.
  - Elles ne sont pas initialisées par défaut.
  - Leur emplacement mémoire est libéré par exemple à la fin de l'exécution d'une fonction secondaire
  - Zone mémoire: **Segment de pile**

DURÉE DE VIE DES VARIABLES (2)

- **Portée de variables**: Portion du programme dans laquelle elles sont définies: variables globales et variables locales.

- **Variables globales**: Variable déclarée en dehors de toute fonction.
  - Elles sont connues par le compilateur dans toute la portion de code qui suit leur déclaration.
  - Elles sont systématiquement permanentes.

DURÉE DE VIE DES VARIABLES (3)

- **Variables locales**: Variable déclarée à l'intérieur d'une fonction du programme
  - Par défaut, elles sont temporaires:
    - Quand une fonction est appelée, elle place ses variables locales dans la pile.
    - A la sortie de la fonction, les variables locales sont dépilées donc perdues.
  - Les variables locales à une fonction ont une durée de vie limitée à une seule exécution de cette fonction.
  - Les valeurs ne sont pas conservées d'un appel au suivant.

DURÉE DE VIE DES VARIABLES (4)

- **Exemple**

```
int n = 10;
void fonction();
void fonction()
{
int n = 0;
n++;
printf("appel numero %d\n",n);
return;
}
void main()
{
int i;
for (i = 0; i < 5; i++)
fonction();
}
```

TRANSMISSION DES PARAMÈTRES D'UNE FONCTION (1)

- Cas 1: les **paramètres** d'une fonction **transmis par valeurs**

- Traités de la même manière que les variables locales

  - Lors de l'appel de la fonction, les paramètres effectifs sont copiés dans le segment de pile.
  - La fonction travaille alors uniquement sur cette copie.
  - Cette copie disparaît lors du retour à la fonction appelante.

  ⇒si la fonction modifie la valeur d'un de ses paramètres, seule la copie sera modifiée;

  ⇒la variable de la fonction appelante ne sera pas modifiée.

TRANSMISSION DES PARAMÈTRES D'UNE FONCTION (2)

```
void echange (int, int );
void echange (int a, int b)
{
int t;
printf("debut fonction :\n a = %d \t b = %d\n",a,b);
t = a;
a = b;
b = t;
printf("fin fonction :\n a = %d \t b = %d\n",a,b);
return;
}
void main()
{
int a = 2, b = 5;
printf("debut programme principal :\n a = %d \t b = %d\n",a,b);
echange(a,b);
printf("fin programme principal :\n a = %d \t b = %d\n",a,b);
}
```

TRANSMISSION DES PARAMÈTRES D'UNE FONCTION (3)

Cas 2: les **paramètres** d'une fonction **transmis par adresse**

Pour qu'une fonction modifie la valeur de l'un de ses arguments, il faut qu'elle ait pour paramètre l'adresse de cette variable et non sa valeur.

TRANSMISSION DES PARAMÈTRES D'UNE FONCTION (2)

Exemple:

```
#include<stdio.h>
void echange (int* a, int* b){
int t;
t = *a;
*a = *b;
*b = t;
}
int main(){
int a = 2, b = 5
;printf("debut programme principal :\n a = %d \t b = %d\n",a,b);
echange(&a,&b);
printf("fin programme principal :\n a = %d \t b = %d\n",a,b);
return 0;
}
```

TABLEAUX

- Définition: Ensemble fini d'éléments de même type, stockés en mémoire à des adresses contigües.

- La déclaration d'un tableau:

  type nom-du-tableau[nombre_éléments];

Où *nombre_éléments* est une expression constante entière positive.

- Par exemple: int tab[10];
  - Indique que tab est un tableau de 10 éléments de type int
  - Alloue un espace de 10*4 octets consécutifs

TABLEAUX

TABLEAUX

- Pour plus de clarté, il est recommandé de donner un nom à la constante *nombre_éléments* par une directive:

  # define *nombre_éléments* 10

- On accède à un élément du tableau en lui appliquant l'opérateur [].
- Les éléments d'un tableau sont toujours numérotés de 0 à *nombre_éléments -1*

TABLEAUX

Afficher les éléments d'un tableau:

```
#define N 10
void main()
{
   int tab[N];
   int i; ...
   for (i = 0; i < N; i++)
         printf("tab[%d] = %d\n",i,tab[i]);
}
```

TABLEAUX

- Un tableau correspond en fait à un pointeur vers le premier élément du tableau.

- Ce pointeur est constant. C'est-à-dire, aucune opération globale n'est autorisée sur un tableau.
  - Notamment, un tableau ne peut pas figurer à gauche d'un opérateur d'affectation
  - On ne peut pas écrire: tab1=tab2;
  - Il faudrait écrire:

```
#define N 10
void main()
{
        int tab1[N], tab2[N];
        int i;
        ...
        for (i = 0; i < N; i++)
        tab1[i] = tab2[i];
}
```

TABLEAUX

- On peut initialiser un tableau au moment de sa déclaration:

type nom-du-tableau[N] = {constante_1,constante_2,...,constante_N};

- Exemple:

```
#include<stdio.h>
#define N 4
void main()
{
    int tab[N] = {1, 2, 3, 4};
    int i;
    for (i = 0; i < N; i++)
    printf("tab[%d] = %d\n",i,tab[i]);
}
```

TABLEAUX

- Si le nombre de données dans la liste d'initialisation est inférieur à la dimension du tableau,
  - seuls les premiers éléments sont initialisés
  - Les autres sont mis à zéro

- De la même manière, un tableau de caractères peut être initialisé par une liste de caractères, mais aussi par une chaîne de caractères.

TABLEAUX

- Il est important de noter que le compilateur complète toute chaîne de caractères avec un caractère nul '\0'.

- Il faut donc que le tableau de caractères est au moins un élément de plus que le nombre de caractères de la chaîne de caractères

- Exemple:

```
#define N 8
char tab[N] = "exemple";
void main()
{
    int i;
    for (i = 0; i < N; i++)
    printf("tab[%d] = %c\n",i, tab[i]);
}
```

TABLEAUX

- Il est aussi possible de ne pas spécifier le nombre d'éléments du tableau.

- Par défaut, il correspondra au nombre de constantes dans la liste d'initialisation:

```
char tab[] = "exemple";  /* ici 8*/
void main()
{
    int i;
    printf("Nombre de caracteres du tableau = %d\n",sizeof(tab)/sizeof(char));
}
```

TRANSMISSION DES PARAMÈTRES D'UNE FONCTION

Rappel:

- Un tableau est un pointeur sur le premier élément du tableau.
- Lorsqu'un tableau est transmis comme paramètre à une fonction secondaire, ses éléments sont donc modifiés par la fonction.

Transmission des paramètres par adresse d'une fonction

Exemple de définition de fonction (voir aussi l'explication au tableau):

```
void rechercheMax(int tab[N], int *max1,int *max2){
        int i,k;
*max1=tab[0];
k=0;
*max2=tab[0];
for (i=1;i<N;i++)   {
                    if (*max1<tab[i]))          {*max1=tab[i]); k=i;
                    }           }
        for (i=1;i<N;i++)   {
if (*max2<tab[i] && i!=k)
*max2=tab[i];     }
        // printf("Les max sont %d et %d \n",*max1,*max2);
}
```

Transmission des paramètres par adresse d'une fonction

Exemple d'appel de fonction:

….

```
int main(){

int tab[N], maximum1, maximum2;

remplir(tab);

rechercheMax(tab, &maximum1, &maximum2);

printf(« Voici les deux maximums: %d et %d\n, maximum1,
maximum2);
return 0;
}
```

Transmission des paramètres par adresse d'une fonction

Petit exercice:

Ecrire une fonction echange qui permute le contenu des deux variables entières. Puis, écrire un programme test qui appelle cette fonction et qui affiche le contenu des deux variables permutées.

TABLEAUX MULTI-DIMENSIONNELS

- De la même manière, on peut déclarer un tableau à plusieurs dimensions
- Exemple d'un tableau à deux dimensions:

  type nom-du-tableau[nombre-lignes][nombre-colonnes]
- Il s'agit d'un tableau unidimensionnel dont chaque élément est lui-même un tableau
- Un élément du tableau: tab[i][j]

TABLEAUX MULTI-DIMENSIONNELS

```
#define M 2
#define N 3
void main()
{
    int tab[M][N] = {{1, 2, 3}, {4, 5, 6}};/* Initialisation*/
    int i, j;
   for (i = 0 ; i < M; i++)
   {
       for (j = 0; j < N; j++)
           printf("tab[%d][%d]=%d\n",i,j,tab[i][j]);
    }
}
```

POINTEURS

POINTEURS

- Toute variable manipulée dans un programme est stockée quelque part en mémoire centrale.
- Cette mémoire est constituée d'octets qui sont identifiés de manière univoque par un numéro qu'on appelle *adresse*.
- *Pour retrouver une variable, il suffit donc de connaître l'adresse* du premier octet où elle est stockée

POINTEURS

- Pour des raisons évidentes de lisibilité, on désigne souvent les variables par des identificateurs, et non par leur adresse.
- C'est le compilateur qui fait alors le lien entre l'identificateur d'une variable et son adresse en mémoire.
- Toutefois, il est parfois très pratique de manipuler directement une variable par son adresse.

POINTEURS

- Un pointeur est une variable dont la valeur est égale à l'adresse d'une autre variable. Comme n'importe quelle variable, sa valeur est modifiable.

- On déclare un pointeur par l'instruction : *type *nom-du-pointeur;*

où *type est le type de la variable pointée.*

- Pour un pointeur sur une variable de objet de type char, la valeur donne l'adresse de l'octet où cette variable est stockée.

- Par contre, pour un pointeur sur une variable de type int, la valeur donne l'adresse du premier des 4 octets où la variable est stockée.

POINTEURS

- Exemple

```
int i = 3;
int *p;
p = &i;
```

Variable	Adresse	Valeur
i	4831836000	3
p	4831836004	4831836000

```
void echange (int *, int *);
void echange (int *adr_a, int *adr_b)
{
int t;
t = *adr_a;
*adr_a = *adr_b;
*adr_b = t;
return;
}
void main()
{
int a = 2, b = 5;
printf("debut programme principal :\n a = %d \t b
= %d\n",a,b);
echange(&a,&b);
printf("fin programme principal :\n a = %d \t b =
%d\n",a,b);
}
```

POINTEURS

- L'*opérateur unaire d'indirection * permet d'accéder directement à la valeur de la variable pointée.*
- Ainsi, si p est un pointeur vers un entier i, *p désigne la valeur de i.

- Par exemple, le programme

```
void main()
{
   int i = 3;
   int *p;
   p = &i;
   printf("*p = %d \n",*p);
}
```

Imprime *p=3

POINTEURS

- Nous sommes dans la configuration suivante:

Variable	Adresse	Valeur
i	4831836000	3
p	4831836004	4831836000
*p	4831836000	3

- Toute modification de *p modifie i. Ainsi, si l'on ajoute l'instruction *p = 0; à la fin du programme précédent, la valeur de i devient nulle.

On peut donc dans un programme manipuler `a la fois les objets p et *p. Ces deux manipulations sont très différentes.

POINTEURS

- Comparons par exemple les deux programmes suivants :

```
void main(){
   int i = 3, j = 6;
   int *p1, *p2;
   p1 = &i; p2 = &j;*p1 = *p2;
}
```

Et

```
void main(){
   int i = 3, j = 6;
   int *p1, *p2;
   p1 = &i; p2 = &j; p1 = p2;
}
```

POINTEURS

- Avant la dernière affectation de chacun de ces programmes, on est dans une configuration du type :

Variable	Adresse	Valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

POINTEURS

Après l'affectation *p1 = *p2; du premier programme, on a

Variable	Adresse	Valeur
i	4831836000	6
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

POINTEURS

- Avant la dernière affectation de chacun de ces programmes, on est dans une configuration du type :

Variable	Adresse	Valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836000
p2	4831835992	4831836004

POINTEURS

Par contre, l'affectation p1 = p2 du second programme, conduit `a la situation:

Variable	Adresse	Valeur
i	4831836000	3
j	4831836004	6
p1	4831835984	4831836004
p2	4831835992	4831836004

ARITHMÉTIQUE DES POINTEURS

- La valeur d'un pointeur étant un entier, on peut lui appliquer un certain nombre d'opérateurs arithmétiques classiques.

- Les seules opérations arithmétiques valides sur les pointeurs sont :

– l'addition d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
– la soustraction d'un entier à un pointeur. Le résultat est un pointeur de même type que le pointeur de départ ;
– la différence de deux pointeurs pointant tous deux vers des variables de même type. Le résultat est un entier.

Notons que la somme de deux pointeurs n'est pas autorisée.

ARITHMÉTIQUE DES POINTEURS

- Si i est un entier et p est un pointeur sur une variable de type *type1, l'expression* ***p + i*** *désigne* un **pointeur** sur une variable de type *type1 dont la valeur est égale à la valeur de* ***p incrémentée de*** **i * sizeof(*type1*).**

- *Il en va de même pour la soustraction d'un entier à un pointeur, et pour*

les opérateurs d'incrémentation et de décrémentation ++ et --.

```
void main(){
   int i = 3;
   int *p1, *p2;
   p1 = &i;
   p2 = p1 + 1;
   printf("p1 = %ld \t p2 = %ld\n",p1,p2);
}    affiche p1 = 4831835984 p2 = 4831835988.
```

ARITHMÉTIQUE DES POINTEURS

- Par contre, le même programme avec des pointeurs sur des variables de type double :

```
void main(){
   double i = 3; /* double sur 8 octets*/
   double *p1, *p2;
   p1 = &i;
   p2 = p1 + 1;
   printf("p1 = %ld \t p2 = %ld\n",p1,p2);
} affiche p1 = 4831835984 p2 = 4831835992.
```

ARITHMÉTIQUE DES POINTEURS

- Les opérateurs de comparaison sont également applicables aux pointeurs, `a condition de comparer des pointeurs qui pointent vers des variables de même type.

- L'utilisation des opérations arithmétiques sur les pointeurs est particulièrement utile pour parcourir des tableaux.

```
#define N 5
int tab[5] = {1, 2, 6, 0, 7};
void main(){
   int *p;
   printf("\n ordre croissant:\n");
   for (p = &tab[0]; p <= &tab[N-1]; p++)
   printf(" %d \n",*p); printf("\n ordre decroissant:\n");
   for (p = &tab[N-1]; p >= &tab[0]; p--) printf(" %d \n",*p);
}
```

POINTEUR SUR VOID

- Le pointeur sur void, aussi connu comme pointeur générique, est un type spécial de pointeur qui peut pointer des variables de n'importe quel type.

- Syntaxe: void * pt;

```
void swap(void * a , void * b)

{ void * tmp;
tmp = a;
a = b;
b = tmp; }
```

```
void main(){
int a = 3;
int b = 2;
swap( &a, &b );
printf( "a = %d , b = %d" , a , b );
}
```

CHAINES DE CARACTERES

CHAÎNES DE CARACTÈRES

- Lire et écrire une chaîne de caractères :

```
#include <stdio.h>
void main() {
char nom[20], prenom[20], ville[25];
printf("quelle est votre ville : ");
gets(ville);
printf("Donnez vos nom et prenom : ");
scanf("%s %s", nom, prenom);
printf("Bonjour cher %s %s, qui habitez a ", prenom, nom);
puts(ville);
}
```

MANIPULATION DE CHAÎNES DE CARACTÈRES

```
#include <stdio.h>
void main()
{
   int i;
   char *chaine;
   chaine = "chaine de caracteres";
   for (i = 0; *chaine != '\0'; i++)
   chaine++;
   printf("nombre de caracteres = %d\n",i);
}
```

LONGUEUR DES CHAÎNES DE CARACTÈRES

- Fonction « strlen» de la librairie <string.h> ayant comme paramètre un pointeur sur un objet de type char
- Cette fonction renvoie un entier dont la valeur est égale à la longueur de la chaine de caractères passée en argument (moins le caractère '\0').
- L'utilisation de pointeurs de caractères au lieu de tableaux permet de créer par exemple une chaîne correspondant à la concaténation de deux chaînes de caractères

GESTION DYNAMIQUE DE LA MÉMOIRE

- On peut initialiser un pointeur p par une affectation sur p. Par exemple, on peut affecter à p l'adresse d'une autre variable.
- Il est également possible d'affecter directement une valeur à *p. Mais pour cela, il faut d'abord réserver à *p un espace-m´emoire de taille adéquate.
- L'adresse de cet espace-mémoire sera la valeur de p.

GESTION DYNAMIQUE DE LA MÉMOIRE

- Cette opération consistant à réserver un espace-mémoire pour stocker l'objet pointé s'appelle *allocation dynamique*.
- *Elle se fait en C* par la fonction malloc de la librairie standard stdlib.h.
- Sa syntaxe est : malloc(*nombre-octets)*

GESTION DYNAMIQUE DE LA MÉMOIRE

- malloc retourne un pointeur de type char * pointant vers un objet de taille *nombre-octets* octets.
- Pour initialiser des pointeurs vers des objets qui ne sont pas de type char, il faut convertir le type de la sortie de la fonction malloc à l'aide d'un cast.
- L'argument *nombre-octets* est souvent donné à l'aide de la fonction sizeof() qui renvoie le nombre d'octets utilisés pour stocker un objet.

GESTION DYNAMIQUE DE LA MÉMOIRE

Soit le programme suivant

```
#include <stdio.h>
#include <stdlib.h>
void main(){
   int i = 3; int *p;
   printf("valeur de p avant initialisation = %ld\n",p);
   p = (int*)malloc(sizeof(int));
   printf("valeur de p apres initialisation = %ld\n",p);
   *p = i;
   printf("valeur de *p = %d\n",*p);}
```

GESTION DYNAMIQUE DE LA MÉMOIRE

Ce programme définit un pointeur p sur un objet *p de type int, et affecte à

*p la valeur de la variable i. Il imprime `a l´ecran :

valeur de p avant initialisation = 0
valeur de p apres initialisation = 5368711424
valeur de *p = 3

Avant l'allocation dynamique, on se trouve dans la configuration

objet	adresse	valeur
i	4831836000	3
p	4831836004	0

GESTION DYNAMIQUE DE LA MÉMOIRE

- A ce stade, *p n'a aucun sens. En particulier, toute manipulation de la variable *p générerait une violation mémoire, détectable à l'exécution par le message d'erreur « Segmentation fault ».

- L'allocation dynamique (malloc) a pour résultat d'attribuer une valeur à p et de réserver à cette adresse un espace-mémoire composé de 4 octets pour stocker la valeur de *p.

GESTION DYNAMIQUE DE LA MÉMOIRE

On a alors

objet	adresse	valeur
i	4831836000	3
p	4831836004	5368711424
*p	5368711424	? (int)

*p est maintenant définie mais sa valeur n'est pas initialisée. Cela signifie que

*p peut valoir n'importe quel entier (celui qui se trouvait précédemment à

cette adresse).

GESTION DYNAMIQUE DE LA MÉMOIRE

- L'affectation *p= i; a enfin pour résultat d'affecter à *p la valeur de i. A la fin du programme, on a donc

objet	adresse	valeur
i	4831836000	3
p	4831836004	5368711424
*p	5368711424	3

GESTION DYNAMIQUE DE LA MÉMOIRE

- Il est important de comparer le programme précédent avec

```
void main(){
   int i = 3; int *p;
   p = &i;}
```

qui correspond à la situation

objet	adresse	valeur
i	4831836000	3
p	4831836004	4831836000
*p	4831836000	3

- Dans ce dernier cas, les variables i et *p sont identiques (elles ont la même adresse) ce qui implique que toute modification de l'une modifie l'autre.

GESTION DYNAMIQUE DE LA MÉMOIRE

- Ceci n'était pas vrai dans l'exemple précédent où *p et i avaient la même valeur mais des adresses différentes.

- On remarquera que le dernier programme ne nécessite pas d'allocation dynamique puisque l'espace-mémoire à l'adresse &i est déjà réservé pour un entier.

GESTION DYNAMIQUE DE LA MÉMOIRE

- La fonction malloc permet également d'allouer un espace pour plusieurs objets contigus en mémoire. On peut écrire par exemple

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

 void main(){
    int i = 3;
    int j = 6;
    int *p;
    p = (int*)malloc(2 * sizeof(int));
    *p = i;
    *(p + 1) = j;
    printf("p = %ld \t *p = %d \t p+1 = %ld \t *(p+1) = %d \n",p,*p,p+1,*(p+1));
}
```

GESTION DYNAMIQUE DE LA MÉMOIRE

- On a ainsi réservé, à l'adresse donnée par la valeur de p, 8 octets en mémoire, qui permettent de stocker 2 objets de type int.

- Le programme affiche

p = 5368711424 *p = 3   p+1 = 5368711428   *(p+1) = 6 .

GESTION DYNAMIQUE DE LA MÉMOIRE

- La fonction calloc de la librairie stdlib.h a le même rôle que la fonction malloc mais elle initialise en plus l'objet pointé *p à zéro.
- Sa syntaxe est calloc(*nb-objets,taille-objets)*

Ainsi, si p est de type int*, l'instruction p = (int*)calloc(N,sizeof(int)); est

Strictement équivalente à:

```
p = (int*)malloc(N * sizeof(int));
for (i = 0; i < N; i++)
*(p + i) = 0;
```

L'emploi de calloc est simplement plus rapide.

GESTION DYNAMIQUE DE LA MÉMOIRE

- Enfin, lorsque l'on n'a plus besoin de l'espace-mémoire alloué dynamiquement (c'est-à-dire quand on n'utilise plus le pointeur p), il faut libérer cette place en mémoire.

- Ceci se fait à l'aide de l'instruction free qui a pour syntaxe : free(*nom-du-pointeur);*

- A toute instruction de type malloc ou calloc doit être associée une instruction de type free.

GESTION DYNAMIQUE DE LA MÉMOIRE

- "realloc" change la taille d'un bloc mémoire précédemment alloué via "malloc" or "calloc". Logiquement "realloc" se comporte comme suit :

```
newp = malloc(size);
if (NULL != newp) { memcpy(newp, oldp, min(size, oldsize)); free(oldp); }
return(newp);
```