III.
Entrées/sorties simplifiées

1. Ecrire un message
2. Lire/écrire un caractère :
3. Lire/écrire un entier :
4. Lire/écrire un réel :
5. Ecrire plusieurs variables dans un message :

Deux fonctions pour lire une variable et l'écrire

Écriture : valeur en binaire => écran (représentation graphique)

`printf`

Lecture : clavier (suite de touches) => variable (en binaire)

`scanf`

Remarque : la lecture est une forme d'affectation.

1) Ecrire un message (suite de caractères)

```
printf("Bonjour ") ;
printf("à tous !\n") ;
```

- '\n' représente la touche <*entrée*>
- l'écriture de '\n' génère un retour à la ligne

2) Lire/écrire un caractère :

```
char car ;
car = 'A';
printf("%c", car) ;
printf("%d", car) ;

scanf("%c", &car) ;
```

affiche **A** *à l'écran*

affiche **65** *à l'écran (code ASCII)*

ou `car = getchar() ;`

Rappel : les petits entiers et les caractères ne se distinguent que par leur convention de lecture/écriture

3) Lire/écrire un entier :

```
int i ;
scanf(“%d”, &i) ;
printf(“%d”, i) ;
```

4) Lire/écrire un réel :

```
float r ;
scanf(“%f”, &r) ;
printf(“%f”, r) ;
```

5) Ecrire plusieurs variables dans un message :

```
int i ;
float x;
...
printf(“resultat : %d et %f\n”, i, x) ;
```

IV.
Expressions

1. Syntaxe d'une expression
2. Opérateurs et expressions arithmétiques
3. Opérateurs et expressions logiques
4. Priorités entre opérateurs
5. Opérateur d’affectation

1. Syntaxe d'une expression

expression

- -> *expression* ***opérateurBinaire*** *expression*
- -> ***opérateurUnaire*** *expression*
- -> **(** *expression* **)**
- -> *variable*
- -> *constante*

2. Opérateurs et expressions arithmétiques

+ - *

/ *division réelle ou entière* (danger !!!)

() *factorisation*

% *modulo*

- *moins unaire (ex : -2)*

Ex : 1 + 3/2 - 5*(1 + 10%3)

Remarque : priorité classique entre les opérateurs

3. Opérateurs et expressions logiques

- En logique, deux valeurs possibles : **vrai** ou **faux**.
- En langage C
  - **0** représente **faux**
  - tout ce qui n'est pas nul représente **vrai**

- opérateurs de comparaison :

==	égal **?**	*danger de confusion avec l'affectation !*
!=	différent **?**	
< et **<=**	inférieur, inférieur ou égal **?**	
> et **>=**	supérieur, supérieur ou égal **?**	

- opérateurs logiques :

!	négation
&&	ET
||	OU

4. Priorités entre opérateurs

- *J'aime les voitures rouges et bleue ou verte ?*
- `i==1 || j<2 ?`

15	( )	->
14	!	<-
13	* / %	->
12	+ -	->
10	< <= > >=	->
9	== !=	->
5	&&	->
4	||	->
2	=	<-

5. Opérateur d'affectation

Syntaxe simplifiée :

affectation

-> *variable* = *expression*

ex : `i = k+1`

1. l'expression `k+1` est évaluée
2. le résultat est copié dans la variable `i`

Remarque : `=` *est un opérateur !!!*

Complément sur la syntaxe de l'affectation :

affectation

`->` *variable* `=` [*variable* `=`] *expression*

Exemple : `i = j = k+1`

1. l'expression *`k+1`* est évaluée
2. `j` reçois cette valeur
3. `i` reçois cette valeur

(*`j = k+1`*) est une expression qui vaut son membre droit

Remarque : associativité droite-gauche (`<-`)

a = b = c = expression <=> *a = (b = (c = expression))*

V.
Syntaxe des instructions

1. Instructions séquentielles
2. Instructions alternatives
3. Instructions répétitives

Trois catégories d'instructions en programmation structurée :

1. séquentielles
2. alternatives
3. répétitives

1. Les instructions séquentielles

instruction-vide :

-> ;

instruction-expression :

-> *expression* ;

instruction-bloc (ou action complexe) :

-> { [*définition*]
[*instruction*]
}

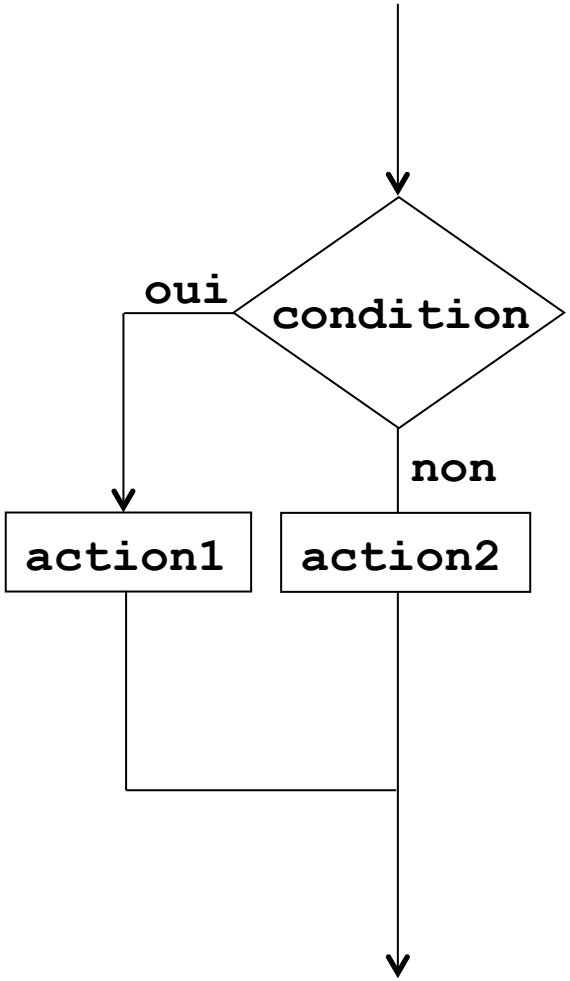
Remarques :

- instruction d'affectation `i = j+1 ;`
- le ‘ ; ’ est un « *terminateur* » pour certaines instructions

2. Les instructions alternatives

```
si (condition)
   action1
fin si
```

```
si (condition)
   action1
sinon
   action2
fin si
```



instruction-if :

if (*expression*)
une instruction
else
une instruction
// fin si

facultatif

Exemples :

- *Faire une remise de 10% au delà de 5 articles vendus*
  - *déclarer les variables*
  - *lire le prix unitaire et le nbre d'articles*
  - *calculer le total*
  - *appliquer la réduction*
  - *afficher le prix final*
- *Résoudre une équation du second degré*

Algorithme de résolution d'une équation du seconde degré

(le cas a=0 ne sera pas traité)

```
soient les réels a, b, c, delta, x1, x2 ;
lire (a, b, c) ;
delta <- b² - 4 a c ;
si (delta <0)
    pas de solution dans R
sinon
    si (delta = 0)
        x1 <-  -b / 2a ;
        écrire (x1) ;
    sinon
        delta <- delta^1/2 ;
        x1 <-  (-b - delta) / 2a ;
        x2 <-  (-b + delta) / 2a ;
        écrire (x1, x2) ;
    fin si
fin si
```

En Résumé

- Instruction alternative simple

```
if(condition)
{ suite d'instructions
}
```

- Instruction alternative à 2 voies

```
if(condition)
{ suite d'instructions
}
else
{ suite d'instructions
}
```

Erreurs courantes :

- Mauvaise indentation
- Plusieurs instructions sans { }
- `if (condition) ;`

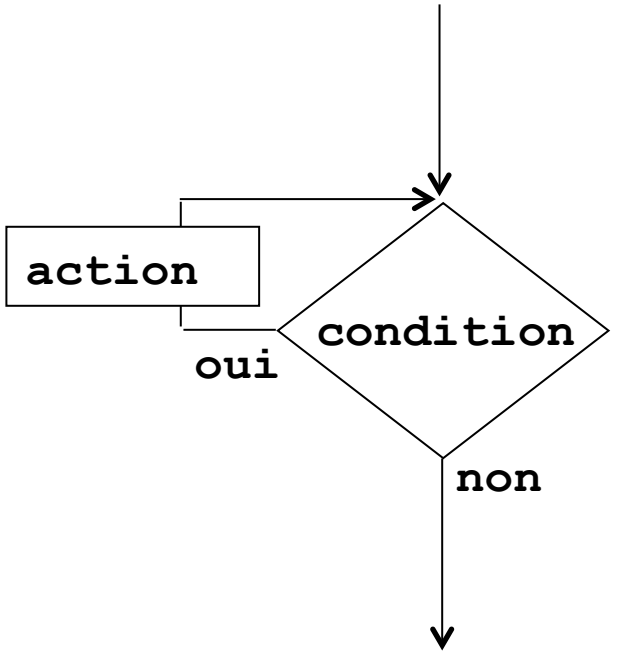
- Instruction alternative à voies multiples

```
if(condition)
{ suite d'instructions
}
else if (condition)
{ suite d'instructions
}
else if (condition)
{ suite d'instructions
}
else
{ suite d'instructions
}
```

3. Les instructions répétitives

- **La boucle *tant-que***

tant-que (*condition*)
    action
fin tant-que



instruction tant-que :

while (*expression*)
    une instruction
// fin tant-que

Deux types de boucle tant-que

- Le nbre d'itérations est connu avant l'exécution de la boucle
- On ne peut pas prévoir le nbre d'itérations

Exemple avec un nombre d'itérations connu : *calculer* x^n

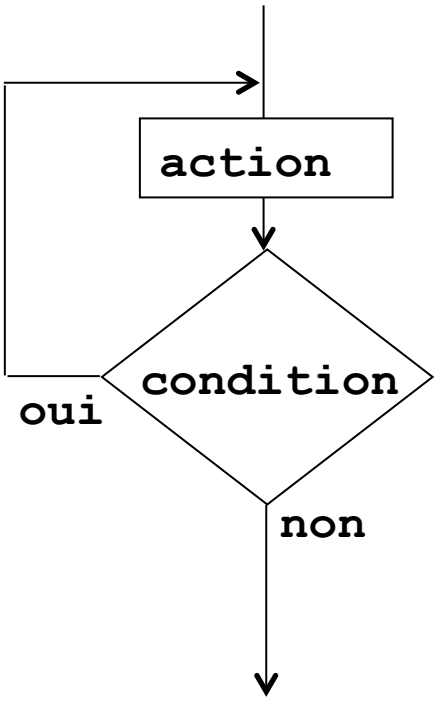
Une telle boucle est constituée :

1. d'une première partie **d'initialisation** (les « **préliminaires** »…)
2. d'une **condition** de répétition
3. d'une **action** à exécuter autant de fois que la condition est vérifiée.

Exemple avec un nombre d'itérations non prévisible :

scanf("%d", &i) ou encore *Algorithme de Horner*

- **La boucle *répéter-tant-que***




```
répéter
   action
tant-que (condition)
```

Remarque : passage obligatoire sur l'action avant le 1er test

instruction-répéter-tant-que

```
do
   une instruction
while (expression) ;
```

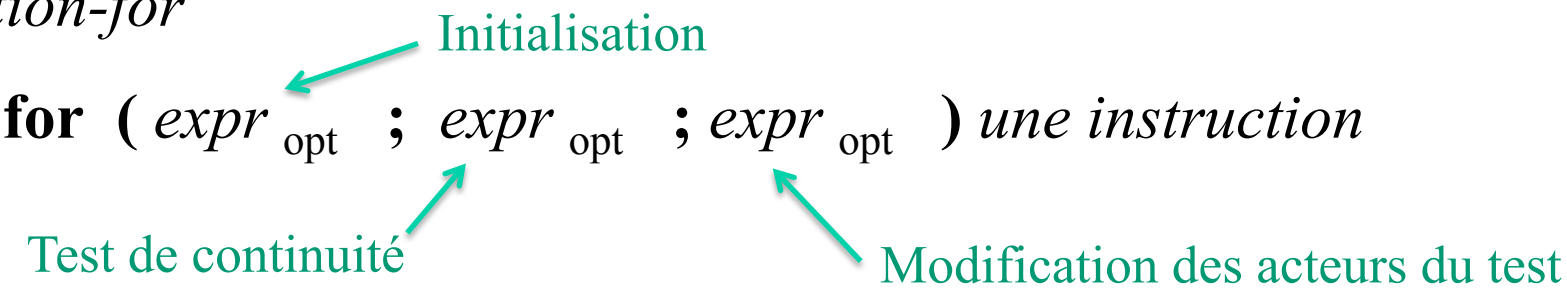
Exemple : *lire le prochain caractère "non blanc"*

- **La boucle *pour***

une variante de la boucle *tant que*

utilisée de préférence quand on connaît le nombre d'itérations

instruction-for



- L'initialisation est effectuée une fois avant de commencer la boucle
- chaque itération lance successivement :
  - l'évaluation du test (abandon de la boucle si **FAUX**),
  - l'instruction (action),
  - la modification des acteurs du test

Exemple: calculer n! avec les boucles ***tant-que*** et ***pour***

En Résumé

- Schéma général d'une boucle avec un nombre N d'itérations connu

```
int cpt ;
cpt = 0; /* nbre d'iterations faites */
while (cpt < N)
{ suite d'instructions
  cpt=cpt+1 ;
}
/* cpt vaut N en sortant de la boucle */
```

Ou encore :

```
cpt = N; /* nbre d'iterations à faire */
while (cpt > 0)
{ suite d'instructions
  cpt=cpt-1 ;
}
/* cpt vaut 0 en sortant de la boucle */
```

En Résumé

- Schéma général d'une boucle avec un nombre N d'itérations connu version boucle **`for`**

```
int cpt ; /* nbre d'iterations faites */
for(cpt=0 ; cpt < N ; cpt=cpt+1)
{ suite d'instructions
}
/* cpt vaut N en sortant de la boucle */
```

En Résumé

- Schéma général d'une boucle avec un nombre d'itérations inconnu au départ

```
Initialiser les variables de ctrle et de calcul;
while (les variables de ctrle respectent un critère)
{ instruction
  …                          ← modification d'au moins
  instruction                  une variable de contrôle
}
```

Erreurs courantes :

- Mauvaise indentation
- `while(condition) ;`
- Variables mal initialisées
- Confusion entre le critère de continuité et le critère d'arrêt

Conseil : tester « manuellement » le début et la fin de la boucle