



ECE PARIS • LYON
ÉCOLE D'INGÉNIEURS

Algorithmique et programmation structurée en C

Cours n°3

Langage C Tests et boucles

Antoine Hintzy

antoine.hintzy@ece.fr



Plan du cours

1. Logique

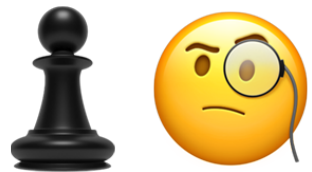
2. Langage C

- Tests
- Boucles

Théorie

Avant de commencer à coder...

Un peu de logique



Théorie

Booléens - Valeurs de vérité {VRAI, FAUX}

	Notation 1 (français)	Notation 2 (entier)	Notation 3 (constantes logiques)
✓	VRAI	1	\top (“Top”)
✗	FAUX	0	\perp (“Bottom”)

En informatique, est considéré comme vrai tout ce qui n'est pas nul (= 0).

Opérateurs logiques - Conjonction, disjonction et négation

	Opérateur	Symbole logique	Opérateur en C/C++	Nombre d'opérandes	Exemples
Conjonction	ET/AND	\wedge ou $.$	<code>&&</code>	2	$A \wedge B, A. B, A\&\&B$
Disjonction	OU/OR	\vee ou $+$	<code> </code>	2	$A \vee B, A + B, A B$
Négation	NON	$\neg a$ ou \bar{a}	<code>!</code>	1	$\neg A, \neg(A \wedge B), \bar{A}, !A$

Avec A et B valant soit *vrai*, soit *faux*.

Tables de vérité des opérateurs logiques ET, OU et NON

		CONJONCTION	DISJONCTION	NÉGATION
		ssi A est vrai et B est vrai	ssi A est vrai ou B est vrai	ssi A est faux
A	B	$A \wedge B$	$A \vee B$	$\neg A$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Théorie

Tables de vérité - Exemple

A	B	$A \wedge B$	$A \vee B$	$A \wedge (A \vee B)$	$A \vee (A \wedge B)$
0	0	0	0	0	0
0	1	0	1	0	0
1	0	0	1	1	1
1	1	1	1	1	1
				$\equiv A$	$\equiv A$

On remarque que l'on peut identifier des équivalences... (*slide* suivante).

Théorie

Équivalences remarquables

$$\neg \top \equiv \perp$$

$$\neg \perp \equiv \top$$

$$\top \wedge A \equiv A$$

$$\perp \vee A \equiv A$$

$$\top \vee A \equiv \top$$

$$\perp \wedge A \equiv \perp$$

$$A \vee \neg A \equiv \top$$

$$A \wedge \neg A \equiv \perp$$

$$\neg \neg A \equiv A$$

$$A \vee A \equiv A \wedge A \equiv A$$

$$A \vee B \equiv B \vee A$$

$$A \wedge B \equiv B \wedge A$$

$$(A \vee B) \vee C \equiv A \vee (B \vee C) \equiv A \vee B \vee C$$

$$(A \wedge B) \wedge C \equiv A \wedge (B \wedge C) \equiv A \wedge B \wedge C$$

$$A \wedge (A \vee B) \equiv A$$

$$A \vee (A \wedge B) \equiv A$$

$$\neg(A \wedge B) \equiv \neg A \vee \neg B$$

$$\neg(A \vee B) \equiv \neg A \wedge \neg B$$

$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$$

$$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$$

Théorie

Exercice - Table de vérité

Remplissez la table de vérité suivante :

A	B	$(A \vee B) \wedge (B \vee (A \wedge B))$
0	0	
0	1	
1	0	
1	1	

Théorie

Exercice - Table de vérité

Aide : on peut décomposer...

A	B	$A \vee B$	$A \wedge B$	$B \vee (A \wedge B)$	$(A \vee B) \wedge (B \vee (A \wedge B))$
0	0				
0	1				
1	0				
1	1				

Ou utiliser les équivalences vues précédemment : $A \vee (A \wedge B) \equiv A$ et $A \vee B \equiv B \vee A$.

Théorie

Exercice - Table de vérité

Correction

A	B	$A \vee B$	$A \wedge B$	$B \vee (A \wedge B)$	$(A \vee B) \wedge (B \vee (A \wedge B))$
0	0	0			
0	1	1			
1	0	1			
1	1	1			

Théorie

Exercice - Table de vérité

Correction

A	B	$A \vee B$	$A \wedge B$	$B \vee (A \wedge B)$	$(A \vee B) \wedge (B \vee (A \wedge B))$
0	0	0	0		
0	1	1	0		
1	0	1	0		
1	1	1	1		

Théorie

Exercice - Table de vérité

Correction

A	B	$A \vee B$	$A \wedge B$	$B \vee (A \wedge B)$	$(A \vee B) \wedge (B \vee (A \wedge B))$
0	0	0	0	0	
0	1	1	0	1	
1	0	1	0	0	
1	1	1	1	1	

Théorie

Exercice - Table de vérité

Correction

A	B	$A \vee B$	$A \wedge B$	$B \vee (A \wedge B)$	$(A \vee B) \wedge (B \vee (A \wedge B))$
0	0	0	0	0	0
0	1	1	0	1	1
1	0	1	0	0	0
1	1	1	1	1	1

Théorie

Mais que seront *A* et *B* dans nos programmes ?

Ce sont soit :

- des valeurs (booléennes)

Exemples : **1** et tout ce qui n'est pas **0** (**2**, **-3**...) sont considérés comme **vrai** et **0** est considéré comme **faux**.

NB : En C++ ainsi que dans d'autres langages, le type booléen existe avec comme valeurs : **true/false**.

- des comparaisons, calculs, fonctions... retournant un booléen.

Exemple : la comparaison (**age** **>=** **18**), qui retourne soit **vrai**, soit **faux**, en fonction de la valeur de **age**.

Fin de la théorie

Revenons au C



Les opérateurs de comparaison en C

Chacun de ces opérateurs retourne **vrai (1)** ou **faux (0)**.

Opérateur	Symbole en C	Exemple en C	Résultat (booléen)
Égalité	<code>==</code>	<code>cpt == 3</code>	<code>1</code>
Strictement inférieur	<code><</code>	<code>3 < 3</code>	<code>0</code>
Strictement supérieur	<code>></code>	<code>age > 16</code>	<code>1</code>
Inférieur ou égal	<code><=</code>	<code>3 <= 3</code>	<code>1</code>
Supérieur ou égal	<code>>=</code>	<code>age >= 18</code>	<code>1</code>
Différent	<code>!=</code>	<code>3 != 3</code>	<code>0</code>

Les opérateurs de comparaison en C

⚠ Ne pas confondre :

- ⦿ l'opérateur de **comparaison** '==' retournant **1** (*vrai*) ou **0** (*faux*).
- ⦿ l'opérateur d'**affectation** '=' donnant une valeur à une variable.

Exemple :

```
int continuer = 1;
```

```
while (continuer = 1) { // ✗ boucle infinie, il faut écrire == pour comparer  
    printf("Voulez-vous continuer ? Saisissez 1 pour continuer, 0 sinon.\n");  
    scanf("%d", &continuer);  
}
```

Priorité des opérateurs

Rappel

Du plus prioritaire au moins prioritaire.

💡 En cas de doute, utilisez des parenthèses :

```
rang < 5 || observations == 0 && bonus >= 1.5
```

// est équivalent à :

```
rang < 5 || (observations == 0 && bonus >= 1.5)
```

// mais pas à :

```
(rang < 5 || observations == 0) && bonus >= 1.5
```

opérateur	associativité
() [] -> .	de gauche à droite
! - ++ -- ~ (type) * & sizeof	de droite à gauche
* / %	de gauche à droite
+ -	de gauche à droite
<< >>	de gauche à droite
< <= > >=	de gauche à droite
= = !=	de gauche à droite
&	de gauche à droite
^	de gauche à droite
	de gauche à droite
&&	de gauche à droite
	de gauche à droite
? :	de droite à gauche
= += -= *= /= >>= <<= &= ^= =	de droite à gauche
,	de gauche à droite

Exercice - Comparaisons en C

Soient deux variables dans un programme C :

```
int var1 = 5,  
    var2 = 9;
```

Que valent les expressions suivantes ? (vrai ou faux)

- `var1 == 5` ?
- `!((var1 % 3) == 0)` ?
- `(var1 < var2) && (var1 <= 0)` ?
- `(var1 > var2) && (var1 >= 0)` ?
- `(var1 > var2) || (var1 >= 0)` ?
- `(var1 < var2) || (var1 <= 0)` ?

Exercice - Comparaisons en C

Correction

Soient deux variables dans un programme C :

```
int var1 = 5,  
    var2 = 9;
```

⦿ `var1 == 5` ?

✅ `var1` vaut 5, 5 est bien égal à 5, donc l'expression est **vraie**.

Exercice - Comparaisons en C

Correction

Soient deux variables dans un programme C :

```
int var1 = 5,  
    var2 = 9;
```

⦿ `!((var1 % 3) == 0) ?`

✅ **var1** vaut 5, **var1 % 3** vaut donc 2, **(var1 % 3) == 0** est donc faux (**2 == 3** est faux), et la négation de faux est vrai. Donc l'expression est **vraie**.

Exercice - Comparaisons en C

Correction

Soient deux variables dans un programme C :

```
int var1 = 5,  
    var2 = 9;
```

⦿ `(var1 < var2) && (var1 <= 0) ?`

✗ `var1 < var2` est vrai car **5** est strictement inférieur à **9**. Comme il s'agit d'une conjonction (ET), pour savoir si l'expression est vraie, l'ordinateur doit évaluer `(var1 <= 0)` qui est faux (car **5** est supérieur à **0**). L'expression est donc **fausse**.

Exercice - Comparaisons en C

Correction

Soient deux variables dans un programme C :

```
int var1 = 5,  
    var2 = 9;
```

⦿ `(var1 > var2) && (var1 >= 0) ?`

✗ `var1 > var2` est faux car **5** n'est pas strictement supérieur à **9**.

Comme nous avons une conjonction (ET), et que la première partie est fausse, alors l'expression est forcément **fausse**.

L'ordinateur n'exécutera/évaluera même pas `(var1 >= 0)` (qui est pourtant vrai).

Exercice - Comparaisons en C

Correction

Soient deux variables dans un programme C :

```
int var1 = 5,  
    var2 = 9;
```

⦿ `(var1 > var2) || (var1 >= 0) ?`

✅ `var1 > var2` est faux car **5** n'est pas strictement supérieur à **9**. Comme il s'agit d'une disjonction (OU), si la seconde partie de l'expression est vraie alors l'expression sera vraie. L'ordinateur évalue la seconde partie. `(var1 >= 0)` est vrai. Donc l'expression est **vraie**.

Exercice - Comparaisons en C

Correction

Soient deux variables dans un programme C :

```
int var1 = 5,  
    var2 = 9;
```

⦿ `(var1 < var2) || (var1 <= 0) ?`

✅ `var1 < var2` est vrai car **5** est strictement inférieur à **9**.

Comme on a une disjonction (OU) et que la première partie est vraie, l'ordinateur n'évalue même pas la seconde partie. L'expression est **vraie**.

Conjonction et disjonction - Ordre

Comme nous venons de le remarquer, l'ordre des opérandes autour des opérateurs **&&** et **||** est important :

- **Conjonction : A && B**

- si **A** est faux, alors B ne sera pas exécuté/évalué, le test est faux.
- si **A** est vrai, alors l'ordinateur doit exécuter/évaluer **B** pour savoir si le test est vrai ou faux.

- **Disjonction : A || B**

- si **A** est vrai, alors B ne sera pas exécuté/évalué, le test est vrai.
- si **A** est faux, alors l'ordinateur doit exécuter/évaluer **B** pour savoir si le test est vrai ou faux.

Utilisons maintenant ces comparaisons dans des tests

Sous la forme de **conditions**

Tests

SI Condition(s) **ALORS** Bloc d'instructions **FIN SI**

```
if ( /* Condition(s) */)
{
    // Bloc d'instructions
}
```

NB : les accolades sont facultatives s'il n'y a qu'une seule instruction. **Il est toutefois conseillé de toujours les mettre.**

Tests - Exemples

SI Condition(s) ALORS Bloc d'instructions FIN SI

```
int a = 3,  
    b = 6,  
    age = 17,  
    majeur = age >= 18; // 😎 majeur prend la valeur retournée  
                        // par la comparaison (age >= 18), soit 0 (faux)
```

<code>if(a == 3)</code>	<code>{ printf("ok"); }</code>	<code>//</code>	✅ vrai, affiche "ok"
<code>if(!(a == 3))</code>	<code>{ printf("ok"); }</code>	<code>//</code>	❌ faux, n'affiche rien
<code>if(a != 3)</code>	<code>{ printf("ok"); }</code>	<code>//</code>	❌ faux, n'affiche rien. NB : équivalent au précédent
<code>if(a < b)</code>	<code>{ printf("ok"); }</code>	<code>//</code>	✅ vrai, affiche "ok"
<code>if(a < b && a > b)</code>	<code>{ printf("ok"); }</code>	<code>//</code>	❌ faux, n'affiche rien. NB : test illogique
<code>if(10)</code>	<code>{ printf("ok"); }</code>	<code>//</code>	✅ vrai car 10 n'est pas 0, affiche "ok"
<code>if(majeur)</code>	<code>{ printf("ok"); }</code>	<code>//</code>	❌ faux car majeur vaut 0, n'affiche rien
<code>if(age)</code>	<code>{ printf("ok"); }</code>	<code>//</code>	✅ vrai car age ne vaut pas 0, affiche "ok"

Tests - Exemples

SI Condition(s) ALORS Bloc d'instructions FIN SI

```
#include <stdio.h>

int main()
{
    int age = 18;

    if (age >= 18 && age <= 25) // ou : if (18 <= age && age <= 25)
    {
        printf("Vous bénéficiez du tarif Jeune.\n");
    }

    return 0;
}
```

⚠ **Un opérande ne peut pas être partagé entre plusieurs opérateurs.**

Ainsi, écrire `if(18 <= age <= 25)` revient à écrire `if((18 <= age) <= 25)` et comme `(18 <= age)` retourne `0` ou `1`, le test est toujours vrai car `0` et `1` sont inférieurs à `25`.

Tests - Exemples

SI Condition(s) **ALORS** Bloc d'instructions **FIN SI**

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

```
int main()
```

```
{
```

```
    int age = 65;
```

```
    if (age >= 60) printf("Vous bénéficiez du tarif Senior.\n");
```

```
    return 0;
```

```
}
```

NB : si un test ne contient qu'une seule instruction dans son bloc d'instructions, celle-ci peut s'écrire sans accolades, sur la même ligne ou sur la ligne suivante. Il est toutefois conseillé de toujours mettre les accolades.

Tests

SI Condition(s) **ALORS** Bloc d'instructions
SINON Bloc d'instructions **FIN SI**

```
if ( /* Condition(s) */)
{
    // Bloc d'instructions
}
else
{
    // Bloc d'instructions
}
```

Tests - Exemple

SI Condition(s) **ALORS** Bloc d'instructions
SINON Bloc d'instructions **FIN SI**

```
#include <stdio.h>

int main()
{
    int age = 20;

    if (age <= 25 || age >= 60)
    {
        printf ("Vous bénéficiez d'un tarif réduit.\n");
    }
    else
    {
        printf ("Vous ne bénéficiez d'aucune réduction.\n");
    }

    return 0;
}
```

Tests

SI Condition(s) **ALORS** Bloc d'instructions **SINON** **SI** Condition(s)
ALORS Bloc d'instructions **SINON** Bloc d'instructions **FIN** **SI**

```
if (/* Condition(s) */)
{
    // Bloc d'instructions
}
else if (/* Condition(s) */)
{
    // Bloc d'instructions
}
else { // le else n'est pas obligatoire
    // Bloc d'instructions
}
```

NB : on peut mettre autant de **else if** que l'on veut (0 ou plusieurs), après le **if**. Le **else** n'est pas obligatoire (mais doit toujours venir en dernier). Un **else if/else** n'est évalué que si aucune des conditions précédentes n'est vérifiée (est vraie).

Tests - Exemple

**SI Condition(s) ALORS Bloc d'instructions SINON SI Condition(s)
ALORS Bloc d'instructions SINON Bloc d'instructions FIN SI**

```
#include <stdio.h>

int main()
{
    int age = 20;

    if (age <= 25)
    {
        printf ("Vous bénéficiez du tarif Jeune.\n");
    }
    else if (age >= 60)
    {
        printf ("Vous bénéficiez du tarif Senior.\n");
    }
    else
    {
        printf ("Vous ne bénéficiez d'aucune réduction.\n");
    }

    return 0;
}
```

Tests - Notation ternaire

```
(/* Condition(s) */) ? /* si vrai */ : /* si faux */;
```

Exemple 1

```
int a = 3;

(a == 3) ? printf("La variable a vaut 3.") : printf("La variable a ne vaut pas 3.");

// ou sur plusieurs lignes :

(a == 3)
    ? printf("La variable a vaut 3.")
    : printf("La variable a ne vaut pas 3.");

// sont tous deux équivalents à :

if (a == 3)
{
    printf("La variable a vaut 3.");
}
else
{
    printf("La variable a ne vaut pas 3.");
}
```

Tests - Notation ternaire

(/* Condition(s) */) ? /* si vrai */ : /* si faux */ ;

Exemple 2 - Utilisation d'un test ternaire dans une instruction.

```
int a = 2, objectif = 3;

printf((a == objectif) ? "La variable a vaut %d.\n" : "La variable a ne vaut pas %d.\n", objectif);

// est équivalent à :

if(a == objectif)
{
    printf("La variable a vaut %d.\n", objectif);
}
else
{
    printf("La variable a ne vaut pas %d.\n", objectif);
}
```

Tests - Notation ternaire

```
(/* Condition(s) */) ? /* si vrai */ : /* si faux */;
```

Exemple 3

```
int b = (a == 0) ? 1 : a;
```

// est équivalent à :

```
int b = 0; // Initialisation
if (a == 0)
{
    b = 1;
}
else
{
    b = a;
}
```

Tests - Switch Case

Les tests **switch case** sont utilisés pour effectuer des tests par valeur exacte (==) sur une même variable. Ils remplacent les **if(a == ...){...} else if(a == ...){...} else if(a == ...){...} else{...}**

```
switch (/* variableATester */)
{
    case /* valeur1 */:
    {
        // Bloc d'instructions à exécuter si variableATester vaut valeur1
        break; // on ne continue pas et on sort du switch
    }

    case /* valeur2 */:
    case /* valeur3 */:
    {
        // Bloc d'instructions à exécuter si variableATester vaut valeur2 ou valeur3
        break;
    }
    // etc.

    default: // facultatif
    {
        // Bloc d'instructions à exécuter si aucune des valeurs ci-dessus ne correspond
        break; // ce break est facultatif (car à la fin) mais conseillé
    }
}
```


Tests - Switch Case - Exemple

```
switch (choix)
{
    case 1:
    case 3:
    {
        printf("Vous avez choisi 1 ou 3.\n");
        break;
    }
    case 2:
    {
        printf("Vous avez choisi 2.\n");
        break;
    }
    default:
    {
        printf("Vous n'avez choisi ni 1, ni 2, ni 3.\n");
        break;
    }
}
```

// est équivalent à :

```
if (choix == 1 || choix == 3)
{
    printf("Vous avez choisi 1 ou 3.\n");
}
else if (choix == 2)
{
    printf("Vous avez choisi 2.\n");
}
else
{
    printf("Vous n'avez choisi ni 1, ni 2, ni 3.\n");
}
```

Boucles - Avec nombre de répétitions connu

```
for (/* initialisation */ ; /* Condition pour continuer la boucle */ ; /* instruction à réaliser après chaque itération */)
{
    // Bloc d'instructions répété
}
```

◎ Exemple :

```
int i = 0; // bien déclarer et initialiser au début du programme
```

```
// POUR i ALLANT DE 0 À 9 (10 exclu) PAR PAS DE 1 (i++)
```

```
for (i = 0; i < 10; i++)
{
    // Bloc d'instructions répété 10 fois
    printf("%d\n", i);
}
```

Boucles - Variable d'itération

On appelle la variable utilisée par une boucle pour compter ses itérations la **variable d'itération**.

Par convention, on appelle les variables d'itération **i**, puis **j**, puis **k**...

Exemple :

```
int i = 0, j = 0;
for (i = 0; i < 10; i++)
{
    for (j = 0; j < 20; j++)          // i est déjà en cours d'utilisation, on a donc besoin de j
    {
        printf("i vaut %d et j vaut %d\n", i, j); // instruction répétée 10*20=200 fois.
    }
}

for (i = 0; i < 10; i++)              // i n'est plus en cours d'utilisation, on peut donc la réutiliser sans créer k
{
    // ...
}
```

Boucles - Variable d'itération

NB : Bien que cela soit **déconseillé**, il est possible de déclarer la variable d'itération directement dans la partie d'initialisation de la boucle :

```
for(int i = 0; i < 10; i++)    // réservation d'un espace mémoire à i
{
    // ... (i existe ici)
}                               // libération de i dans la mémoire

// i n'existe plus !

for(int i = 0; i < 15; i++)    // réservation d'un espace mémoire au nouveau i
{
    // ... (le nouveau i existe ici)
}                               // libération de i dans la mémoire
```

En effet, si la variable est utilisée dans plusieurs boucles comme c'est le cas dans cet exemple, celle-ci sera en théorie allouée en mémoire à chaque déclaration, et donc à chaque boucle. Dans la pratique, les compilateurs actuels corrigent automatiquement ce détail. Mais pour bien faire, déclarez et initialisez les itérateurs comme les autres variables, au tout début du bloc.

Boucles - Tant que

```
while ( /* Condition pour continuer */ )  
{  
    // Bloc d'instructions répété  
}
```

// Exemple :

```
int note = 0;  
scanf("%d", &note);  
  
while (note < 0 || note > 20)  
{  
    printf("La note saisie n'est pas valide, recommencez.\n");  
    scanf("%d", &note);  
}  
  
printf("La note est : %d.\n", note);
```

NB : Ici, on ne rentrera même pas dans la boucle si l'utilisateur rentre une note entre 0 et 20 dès le début !

Boucles - Tant que - Mais au moins une fois

```
do
{
    // Bloc d'instructions
}
while ( /* Condition pour continuer */ );
```

// Exemple :

```
int choix = 0, bonNombre = 6;

do
{
    printf("Choisissez un nombre (0 pour arrêter) :");
    scanf("%d", &choix);
}
while (choix != 0 && choix != bonNombre);
```

NB : même si la condition est fausse dès le départ, le bloc d'instructions sera quand même réalisé une fois.

Boucles - Tant que - Mais au moins une fois

```
do
{
    printf("Choisissez un nombre (0 pour arrêter) :");
    scanf("%d", &choix);
}
while (choix != 0 && choix != bonNombre);

if (choix == bonNombre)
{
    printf("Vous avez trouvé !\n");
}

//***** Est équivalent à : *****

printf("Choisissez un nombre (0 pour arrêter) :");
scanf("%d", &choix);

while (choix != 0 && choix != bonNombre)
{
    printf("Choisissez un nombre (0 pour arrêter) :"); // inconvenient : ces deux lignes de code sont dupliquées = ✗ pas bien !
    scanf("%d", &choix);                               // avantage : on pourrait indiquer un message différent ici (après une première erreur)
}

if (choix == bonNombre)
{
    printf("Vous avez trouvé !\n");
}
```

Préparation des exercices

Première utilisation du débogueur

Copiez-collez le code source suivant dans CLion, ajoutez les points d'arrêts ( `int varA, varB = 0;`) demandés et testez le **mode debug** ().
Testez notamment les boutons :  et . **Que font-ils ? Que permet de voir le débogueur ?**

```
#include <stdio.h>
#define MAX_I 4
#define MAX_J 3

int main() {
    int varA, varB = 0;
    int i = 0, j = 0, cpt = 0;
    // Mettre un point d'arrêt à la ligne suivante, et regarder la valeur de varA :
    varA = 0;
    // Mettre un point d'arrêt à la ligne suivante, et regarder la valeur de varA :
    for(i = 1; i < MAX_I; i++) {
        for(j = 1; j < MAX_J; j++) {
            // Mettre un point d'arrêt à la ligne suivante :
            varA = i * j;
            // Mettre un point d'arrêt à la ligne suivante :
            varB = i + j - i;
            // Mettre un point d'arrêt à la ligne suivante :
            printf("Ceci est le %dème printf !\n", ++cpt);
        }
    }
    return 0;
}
```