Introduction au langage C - Partie 2

et protection contre quelques bugs

Nancy-Université
Université Henri Poincaré

Brigitte Wrobel-Dautcourt

Octobre 2007

Département Informatique - Faculté des Sciences
Université Henri Poincaré - Nancy 1

Partie2

7. Les expressions
8. Les entrées-sorties élémentaires ***printf()*** et ***scanf()***
9. Les instructions
10. Les structures de données (structures, unions)

---

7. Les expressions

Le langage C offre un jeu très étendu d'opérateurs, ce qui permet l'écriture d'une grande variété d'expressions. Un principe général est que *toute expression retourne une valeur*. On peut donc utiliser le résultat d'une expression comme partie d'une autre expression. De plus, le parenthésage permet de forcer l'ordre d'évaluation.

Les expressions

Les opérateurs arithmétiques

`+ - * /`

`%` : modulo
`>>` : décalage à droite
`<<` : décalage à gauche
`&` : et logique (bit à bit)
`|` : ou logique (bit à bit)
`^` : ou exclusif (XOR)
`~` : complément à 1 (NOT)

Exemples :

`i = j << 3`

`j = x & 0x00ff` : et logique

`(k | ~0xf0)` est équivalent à `(k | 0x0f)`

Les opérateurs arithmétiques

+ addition
- soustraction
* multiplication
/ division (entière ou réelle)
% modulo (sur les entiers)

Les opérateurs logiques

Ce sont les opérateurs permettant d'effectuer des opérations au niveau des bits (masquages ...) :

- `&` AND, exemple : `a & 0x000F` extrait les 4 bits de poids faible de `a`
- `|` OR, ainsi, `b = b | 0x100` met à 1 le 9ème bit de `b`
- `^` XOR
- `<<` décalage à gauche, `a = b << 2` met dans `a` la valeur de `b` où tous les bits ont été décalés de 2 positions vers la gauche
- `>>` décalage à droite
- `~` complément à 1 (opérateur unaire)

Les opérateurs d'affectation

`=`

- l'affectation est une expression qui possède une valeur

`a = 3 ;`

- son résultat est égal à la valeur finale du membre gauche (ici 6)

`i = j = k = 6 ;`

est équivalent à : i = (j = (k = 6));

Les opérateurs d'affectation (suite)

- simplification d'écriture :

<variable> <op> = <expr>

- quelques opérateurs :

`+= -= *= /= <<= ...`

`x += 3 ;` est équivalent à x = x + 3
`x /= 2. + y ;` est équivalent à x = x / (2. + y)

L'affectation

= affectation

Il faut bien noter que le signe = est l'opérateur d'affectation, et non de comparaison ; cela prête parfois à confusion et entraîne des erreurs difficiles à discerner. À noter aussi que l'affectation est une expression comme une autre, c'est-à-dire qu'elle retourne une valeur. Il est donc possible d'écrire :

`a = b = c + 2`

ceci revenant à affecter à **b** le résultat de l'évaluation de **c+2**, puis à **a** le résultat de l'affectation **b = c+2**, c'est-à-dire la valeur qu'on a donnée à **b**. Remarquez l'ordre d'évaluation de la droite vers la gauche.

Modifier la valeur d'une variable

Il arrive très souvent qu'on calcule la nouvelle valeur d'une variable en fonction de son ancienne valeur. Le langage C fournit pour cela un jeu d'opérateurs combinés, de la forme

<variable> <op>= <expr>

où *<op>* est un opérateur. Une telle expression est équivalente à l'expression :

<variable> = <variable> <op> <expr>

+= : **a += b** équivaut à **a = a + b**

de même pour : **-=** ***=** **/=** **%=** **<<=** **>>=** **&=** **|=** **^=**

Introduction au langage C
└Les expressions

Les opérateurs d'incrémentation et de décrémentation

++ **--**

L'opération a lieu avant ou après l'évaluation de la variable, cela dépend de la position de l'opérateur par rapport à la variable :

- **a++ ;** est équivalent à **a = a + 1 ;**
- **++a ;** est équivalent à **a = a + 1 ;**

69 / 237

Introduction au langage C
└Les expressions

Les opérateurs d'incrémentation et de décrémentation (suite)

- **b = a++ ;** évaluation de l'expression avant d'incrémenter la variable, équivalent à :

```
b = a ;
a = a + 1 ;
```

- **b = ++a ;** évaluation de l'expression après avoir incrémenté la variable, équivalent à :

```
a = a + 1 ;
b = a ;
```

70 / 237

Les opérateurs d'incrémentation et de décrémentation

++ incrémentation
-- décrémentation

Ces opérateurs, qui ne peuvent être appliqués que sur les types scalaires, peuvent s'employer de deux manières : en principe, s'ils préfixent une variable, celle-ci sera incrémentée (ou décrémentée) avant utilisation dans le reste de l'expression ; s'ils la postfixent, elle ne sera modifiée qu'après utilisation. Ainsi :

```
a = 5 ;
b = 6 ;
c = ++a - b ;
```

donnera à **c** la valeur 0, alors que :

```
a = 5 ;
b = 6 ;
c = a++ - b ;
```

lui donnera la valeur -1.

Les opérateurs de comparaison

`==` `!=` `<` `>` `<=` `>=`

- `(x = y)` est toujours vrai quand $y \neq 0$ puisqu'une affectation est une expression à part entière et que son résultat est la valeur du membre de gauche (ici `x` prend la valeur de `y`).

Les opérateurs relationnels

`&&` : et logique
`||` : ou logique
`!` : non logique

Les opérateurs relationnels et de comparaison

`>`	`>=`	`<=`	`<`	comparaisons
`==`	`!=`	égalité et inégalité		
`!`	négation (opérateur unaire)			
`&&`	ET relationnel			
`||`	OU relationnel			

Il n'y a pas de type booléen en C ; on utilise donc des entiers, et usuellement 0 représente *faux* et toute valeur non nulle représente *vrai*.

L'évaluation d'une expression (ordre)

- le langage C garantit l'ordre d'évaluation d'une expression **relationnelle**, de la gauche vers la droite.

Par exemple :

```
expr1 && expr2
```

est évaluée comme suit :

```
si expr1 alors si expr2 alors 1
                        sinon 0
               fsi
         sinon 0
fsi
```


Il est à noter que le langage C *garantit* l'ordre d'évaluation d'une expression relationnelle : de gauche vers la droite. Cela est intéressant dans de nombreux cas, et évite l'écriture de tests plus complexes, nécessaire parfois dans d'autres langages comme PASCAL. On peut illustrer ce point en traduisant en langage algorithmique l'évaluation de deux expressions relationnelles :

```
expr1 || expr2
```

est équivalent à :

```
si expr1 alors 1
         sinon si expr2 alors 1
                        sinon 0
               fsi
fsi
```

L'évaluation d'une expression (ordre) – (suite)

On arrête l'évalutation d'une expression dès que la valeur de vérité est trouvée :
Exemples :

```
(k < tabLength && tab[k] != 0)
```

```
((i <= 5) && ((c = getchar()) != '\n'))
```

```
(f >= tabLength || tab[i] == '/')
```


associativité des opérateurs

Lorsque deux opérateurs ont le même niveau de priorité, il subsiste une ambiguïté.
Par exemple :

```
x - y + 1
```

peut être interprété par :

- `(x - y) + 1`
- `x - (y + 1)`

Tableau récapitulatif des opérateurs

Les opérateurs sont classés dans l'ordre décroissant des priorités, et en indiquant à chaque fois le sens d'évaluation :

1	fonction/sélection	`() [] . ->`	G–D
2	unaire	`* & - ! ~ ++ - sizeof`	D–G
3	multiplicatif	`* / %`	G–D
4	additif	`+ -`	G–D
5	décalages	`<< >>`	G–D
6	relationnels	`< > <= >=`	G–D
7	inégalité/égalité	`== !=`	G–D
8	et logique	`&`	G–D
9	xor logique	`^`	G–D
10	ou logique	`|`	G–D
11	et relationnel	`&&`	G–D
12	ou relationnel	`||`	G–D
13	conditionnel	`? :`	G–D
14	affectation	`= <op>=`	D–G
15	virgule	`,`	G–D

Exemples :

```
while (c = tab[i] != -1)
```

- comparaison de tab[i] à -1
- affectation du résultat de la comparaison (vrai ou faux) à c

```
while ((c = tab[i]) != -1)
```

- affectation de tab[i] à c
- comparaison de c à -1

Considérons cette instruction :

```
while (c = getc(fich) != EOF)
```

L'affectation est de priorité inférieure à l'opérateur de comparaison, aussi cette instruction :

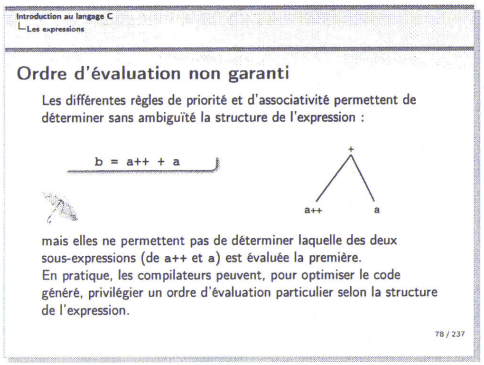
- compare le résultat de la fonction ***getc()*** à la valeur `EOF`, puis
- affecte le résultat de cette expression à la variable `c`.

Il faut ajouter des parenthèses pour changer la priorité des opérateurs et obtenir une écriture correcte :

```
while((c = gets(fich)) != EOF) ...
```

Cette instruction :

- affecte le résultat de la fonction ***getc()*** à la variable **c**, puis
- compare la valeur de **c** à la constante **EOF**.

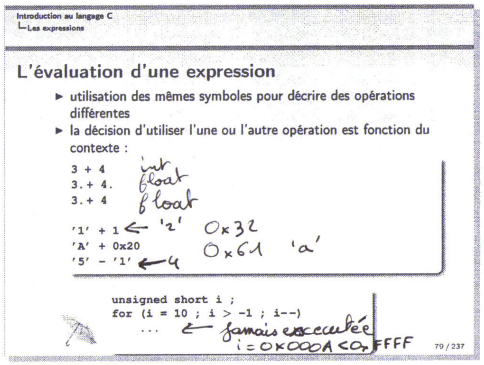


Faites cependant attention dans les expressions un peu complexes où on réutilise la même variable plusieurs fois : l'ordre d'évaluation n'est pas garanti, et l'expression peut donc avoir des résultats différents suivant la machine utilisée.

Ici, les contenus des variables **b** et **i** dépendent du compilateur utilisé :

```
a = 5 ;
b = a++ + a*2 ;

i = 0 ;
fonction(i++, t[i]) ;
```

Les expressions arithmétiques

- **char** et **short** → **int**
- si l'un des opérandes est **double**, les autres sont convertis en **double**,
- sinon si l'un des opérandes est **long**, les autres sont convertis en **long**,
- sinon si l'un des opérandes est **unsigned**, les autres sont convertis en **unsigned**.

Attention : il faut toujours veiller à former des expressions homogènes, en utilisant au besoin la conversion explicite de type **cast** (voir ci-après) et ne jamais tenir compte des éventuelles tolérances du compilateur.

Introduction au langage C
└Les expressions

La conversion explicite de types (cast)

```
(type) expression
```

- type : type primitif : `char`, `int`, `short`, `unsigned char`, ... ou pointeur sur un type quelconque : `int *`, `char *`, ...

```
int h ;
h = (int)(3.9 / 3.)   ;—> 1
```

80 / 237

La conversion de types

On désire souvent changer le type du résultat retourné par une expression. Pour cela existe le mécanisme de *cast* :

(<*nom de type*>) **expression**

qui retourne une valeur dont le type est celui indiqué dans la première parenthèse, et qui est obtenue en convertissant le résultat de l'expression dans le type spécifié.

On peut convertir tout en tout, bien qu'à un niveau sémantique cela puisse parfois être dangereux.

Introduction au langage C
└Les expressions

La conversion explicite de types (cast) (suite)

Exemple :

```
/* déclaration des fonctions */
double sin(double u) ;
double cos(double u) ;

int main(void)
  {
    int x, y ;
    double z ;
    ...
    z = sin((double)(x)) ;
    y = (int)(cos((double)(x)) ;
    ...
```

81 / 237

8. Les entrées-sorties élémentaires

Introduction au langage C
└Les entrées-sorties élémentaires

Les entrées - sorties élémentaires

- le langage C ne possède pas d'instructions d'entrées-sorties
- les entrées-sorties sont réalisées via des **fonctions**
  - lecture par défaut sur l'entrée standard : **stdin** (clavier du terminal)
  - écriture par défaut sur la sortie standard : **stdout** (écran du terminal)
  - inclusion du fichier **stdio.h** contenant la description des profils de certaines fonctions gérant les entrées-sorties.

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

82 / 237

Les entrées/sorties ne font pas partie du langage C proprement dit. Toutefois, il existe une *bibliothèque standard* (*standard I/O library* : `stdio`). Cette bibliothèque définit :

- 3 fichiers standards `stdin`, `stdout` et `stderr`, ouverts implicitement, et correspondant respectivement au flot d'entrée, au flot de sortie et au flot des messages d'erreur,
- une fonction d'écriture sur le fichier `stdout` : ***printf()***,
- une fonction de lecture sur `stdin` : ***scanf()***.

Introduction au langage C
└ Les entrées-sorties élémentaires

L'écriture

l'affichage selon un format sur la sortie standard s'effectue avec la fonction *printf()*, qui admet éventuellement un nombre **variable** de paramètres :

```
int printf(format[, listeVariables])
```

- `format` : chaîne de caractères imprimables et formats d'écriture des variables

  `%d, %f, %.2f, %s, %h, \n, \t, ...`

- `listeVariables` : liste de variables ou d'expressions, séparées par des virgules, dont on veut écrire le contenu

la fonction retourne le nombre de caractères écrits, ou -1 si une erreur s'est produite

84 / 237

Introduction au langage C
└ Les entrées-sorties élémentaires

Exemples :

```
printf("bonjour\n") ;
```

affichera `bonjour`

```
printf("prix : %.2f F à payer avant le %d août\n",px,jour);
```

affichera `prix : 25.50 F à payer avant le 10 août`
(si px et jour contiennent respectivement les valeurs 25.5 et 10)

cases mémoire | 00000000 | 01001011 |

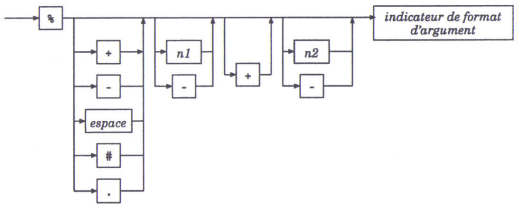
```
printf("%d, %o, %x, %c\n",75,75,75,75)
```

affiche 75, 113, 4b, K

85 / 237

format est une chaîne de caractères composée de caractères imprimables et d'indicateurs de format spécifiant les conversions à effectuer pour l'écriture des arguments.

Un **indicateur de format** commence toujours par le **caractère %** et obéit au modèle suivant :



Les **différents indicateurs de format d'argument** permettent de préciser la conversion à effectuer :

- **d** : entier signé
- **u** : entier non signé
- **o** : octal
- **x** : hexadécimal
- **e** : réel, en notation exponentielle
- **lf** : réel long
- **f** : réel, en notation avec virgule,
- **g** : réel, en notation **e** ou **f** (la plus courte des 2)
- **c** : caractère
- **s** : chaîne de caractères

Exemple :

```
printf("%d : %f, (%.15g, %lf)\n", i, x, alpha, beta) ;
```

i est un entier, **x** est un réel, **alpha** est un réel écrit sur 15 chiffres maximum après le point décimal, **beta** est un réel double.

Signification des drapeaux :

- **-** : cadre l'argument à gauche (par défaut, l'argument est cadré à droite)
- + : fait précéder le nombre de son signe
- *espace* : fait précéder les nombres positifs par un espace
- # : force les nombres réels à être écrits avec un point décimal
- *n1* : écrit l'argument sur *n1* caractères au plus
- *n2* : pour **d, u, o** et **x**, écrit l'argument sur *n2* caractères exactement
  pour **e** et **f**, écrit *n2* chiffres après le point décimal
  pour **g**, écrit au plus *n2* chiffres significatifs
  pour **s**, écrit au plus *n2* caractères
- ***** : remplace *n1* ou *n2*

Caractères d'échappement :

- `\n` : nouvelle ligne
- `\t` : tabulation
- `\f` : nouvelle page
- `\b` : *backspace*
- `\r` : pour ajouter le caractère `^M` si cher à Windows pour gérer les fins de lignes...

...

Syntaxe de la commande

```
printf(format[, arguments])
```

On peut donc utiliser la fontion ***printf()*** avec seulement un format, par exemple :

```
printf("bonjour\n") ;
```

Dans ce cas, le format est considéré comme une constante chaîne de caractères ; ce format peut bien entendu être stocké dans une variable de type chaîne de caractères :

```
{ char chaine[100] ;

  strcpy(chaine,"bonjour\n") ;
  printf(chaine) ;
}
```

qui affichera donc : **bonjour**

On peut également composer des formats "dynamiques" :

```
{ char chaine[100], car ;
  int i ;

  strcpy(chaine,"%c%d\n") ;
  printf(chaine, car, i) ;
}
```


Exemples :

- `%2d` affiche ␣3 si valeur = 3
  `%02d` affiche 03 si valeur = 3
  `%4.2f` affiche 4 caractères pour la partie entière et 2 caractères pour la partie décimale
- le caractère '`%`' doit être doublé pour être affiché.
- le caractère '`\n`' permet de vider le buffer
- le caractère '`\t`' permet d'écrire une tabulation

*Exemple d'utilisation du résultat de la fonction **printf()** :*

```
#include <stdio.h>

int main()
{
  char nom[100] ;
  int nb ;

  scanf("%s", nom) ;

  nb = printf("%s", nom) ;

  if (nb > 20)
    printf("\n") ;

  return(0) ;
}
```


La lecture

```
int scanf(format, listeAdressesVariables)
```

- **format** : chaîne de caractères et formats de lecture des variables
  `%d` `%f` `%c` `%s` ...
- **listeAdressesVariables** :
  adresses séparées par des virgules,
  &variable, ou nom d'un tableau de caractères, ou nom d'un pointeur, ...
- **résultat** : la fonction retourne le nombre d'arguments lus ou un code d'erreur
  - **0** : quand il y a un problème de conversion
  - **EOF** : quand plus de données à lire

La lecture (suite)

exemple :

```
...
while (scanf("%f", &valeur) != EOF)
      {
        ...
      }
...
```

- EOF est une constante défininie dans le header **stdio.h**
- lecture des données sur l'entrée standard (clavier)
  ^D pour terminer le flux des données
  ^D positionne EOF
- lecture des données à partir d'un fichier
  la fin du fichier positionne EOF

Il faut préciser le format de lecture alors que l'on connaît le type de la variable.
Exemples :

```
int valeur ;
scanf("%d", &valeur) ;
```

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

```
char nom[10] ;
scanf("%s", nom ) ;
```

scanf("%s", &nom[0]);


```
scanf("%d heures %d minutes %d secondes", &h, &m, &s) ;
```

demande d'entrer les valeurs sous la forme :
`23 heures 45 minutes 11 secondes`
et non : `23 h 45 min 11 sec`
Il est possible d'ignorer les chaînes de caractères entre les valeurs par la présence du caractère * dans la spécification du format :

```
scanf("%d %*s %d %*s %d %*s", &h, &m, &s) ;
```

Il est préférable d'effectuer 3 appels successifs à la fonction *scanf()* :

```
scanf("%d",&h) ;
scanf("%d",&m) ;
scanf("%d",&s) ;
```


Attention

1. Les arguments de *scanf()* sont toujours des **adresses** :
   (sauf le premier argument qui est un format...)

   ```
   int x ;
   scanf("%d",x) ;
   ```

   est incorrect !

   il faut écrire : scanf("%d", &x) ;

2. La fonction *scanf()* ne permet pas de lire des chaînes de caractères contenant des espaces :

   ```
   char str[100] ;
   scanf("%s",str) ;
   ```

   Pour cela, on peut utiliser la fonction *gets()* :

   ```
   gets(str) ;
   ```

*Exemple d'utilisation de la fonction **gets()** :*

```
#include <stdio.h>

int main()
{
  char nom[40] ;

  printf("nom : ") ; gets(nom) ;
  printf("nom lu : %s\n", nom) ;
  ...
}
```


La fonction ***gets()*** est présenté page 151.

3. Mélange de lectures de caractères et de valeurs numériques :

```
int x, y ;
char c ;

printf("entier :  ") ; scanf("%d",&x) ;
printf("entier :  ") ; scanf("%d",&y) ;
printf("entiers lus :  %d %d\n", x, y) ;
printf("caractere :"); scanf("%c",&c) ;
printf("caractere lu :  %c\n", c) ;
```

Résultat de l'exécution :

entier : 34
entier : 2
entiers lus : 34 2
caractere : caractere lu :

Solutions :

1. `fflush(stdin) ;` non portable
2. scanf("%c%c", &c,&c)
3. scanf("%c%*c", &c)
4. scanf("\n%c", &c)
5. scanf("%d\n", &x)
   scanf("%d \n", &y)

1. `fflush(stdin) ;` La fonction ***fflush()*** active les entrées-sorties physiques.
2. `scanf("%c%c",&c,&c) ;` Le premier format `%c` correspond à la lecture du caractère '\n' de la fin de l'entrée précédente. Le deuxième format `%c` correspond à la lecture du caractère.
3. `scanf("%*c%c",&c) ;` Le premier format `%*c` correspond à la lecture du caractère '\n' de la fin de l'entrée précédente, ce caractère n'est pas mémorisé. Le deuxième format `%c` correspond à la lecture du caractère.
4. Une quatrième solution à ce problème pourrait être de tout lire sous forme d'une chaîne de caractères (avec la fonction ***gets()***), puis de rechercher les informations dans cette chaîne de caractères :

```
#include <stdio.h>
int main(void)
   { int x, y ;
     char c ;
     char nom[20] ;

     printf("entier :  ") ; gets(nom) ; sscanf(nom,"%d",&x) ;
     printf("entier :  ") ; gets(nom) ; sscanf(nom,"%d",&y) ;
     printf("entiers lus :  %d %d\n", x, y) ;

     printf("caractere :  ") ; gets(nom) ; sscanf(nom,"%c",&c) ;
     printf("caracteres lu :  %c\n", c) ;
     return(0) ;
   }
```

La fonction ***sscanf()*** (présentée page 104) analyse la *chaîne* selon le *format* indiqué et place les résultats dans la *liste des variables* dont on a précisé l'adresse. Syntaxe :

`sscanf(chaîne, format, listeAdressesVariables)`

Les deux dernières lignes du programme précédent :

```
sscanf(nom,"%c",&c) ;
printf("caracteres lu :  %c\n", c) ;
```

pourraient être remplacées par : `c = nom[0] ;` Attention, un mélange d'appel entre les fonctions ***scanf()*** et ***gets()*** provoque des erreurs lors de l'exécution.

Introduction au langage C
└Les entrées-sorties élémentaires

J'utilise (mal) mon programme...

code source :

```
int n ;
printf("nombre : ") ;
scanf("%d",&n) ;
...
```

exécution :

```
nombre : .2
```

96 / 237

9. Les instructions

Introduction au langage C
└Les instructions

Les instructions

- les conditionnelles :
  - branchement conditionnel **if - else**
  - branchement multiple **switch - case**
- les itérations :
  - itération **for**
  - itération **while**
  - itération **do - while**
- les ruptures de séquence :
  - **break**
  - continue
  - goto
- les sorties :
  - **return**
  - **exit**

97 / 237

Chaque instruction est terminée par un point-virgule. À noter que le point-virgule est une *terminaison* d'instruction et non un séparateur d'instruction comme en PASCAL. En particulier, pour qu'une expression soit considérée comme une instruction, elle doit être terminée par un `;` même si elle est la dernière d'un bloc.

Un *bloc* est une suite d'instructions délimitées par une accolade ouvrante `{` et une accolade fermante `}`. À l'intérieur de tout bloc, on peut aussi définir des variables *locales* à ce bloc :

```
if (n >0)
        {
          int i ;

          for (i=0 ; i < n ; i++)
                ...
          ...
        }
```

Introduction au langage C
└Les instructions

Branchement conditionnel : if - else

```
if (expression)
   instruction
```

```
if (expression)
   instruction
else
   instruction
```

- **expression** a un résultat booléen
- **instruction** est soit une instruction simple (se terminant par ;), soit un bloc d'instructions (encadré par `{` et `}`)
- **`else`** se rapporte toujours au **`if`** le plus proche qui précède et pour lequel on n'a pas déjà associé une partie **`else`**.

98 / 237

L'*instruction conditionnelle* s'exprime de la manière suivante :

```
    if (<expression>) <instruction-1>
[   else <instruction-2> ]
```

où l'exécution de la branche *alors* ou de la branche *sinon* va dépendre de l'évaluation de *<expression>* : si le résultat est non nul, on exécutera *<instruction-1>*, sinon on effectuera *<instruction-2>*. De manière tout à fait classique, s'il y a plusieurs instructions dans la partie *alors* ou la partie *sinon*, on écrira un bloc délimité par `{` et `}`.

Quand il y a plusieurs conditions imbriquées et qu'il y a ambiguïté sur un **`else`**, on le rattache au **`if`** le plus proche et pour lequel on n'a pas déjà associé une partie **`else`**.

Exemple :

```
if (d != 0) /* test du dénominateur */
   if (r = (n % d)) /* calcul du reste */
      printf("n%%d soit %d%%%d = %d\n", n, d, r) ;
   else
      printf("le reste est nul\n") ;
```

a un résultat différent de :

```
if (d != 0)
   {
     if (r = (n % d))
        printf("n%%d soit %d%%%d = %d\n", n, d, r) ;
   }
else
   printf("attention, le diviseur est nul\n") ;
```

Le test : est syntaxiquement correct et il est équivalent à :

(a)
```
if (r)
   {
     ...
   }
```

(b)
```
if (r != 0)
   {
     ...
   }
```

Pour des raisons de lisibilité, il est préférable de réserver la forme de test *(a)* aux variables ne pouvant prendre que des valeurs booléennes, à savoir 0 et 1.

Il est possible d'insérer des instructions vides dans le code, mais le compilateur ne détecte pas les instructions `if` inutiles du style :

```
if (x > max) ;
        max = x ;
```

(syntaxiquement correct) qui sera donc interprété comme l'exécution systématique des deux instructions :

L'opérateur ternaire conditionnel

```
expression ? expression_1 : expression_2
```

est évalué par : si (expression)
alors expression_1
sinon expression_2
fsi

Exemples :

```
max = (a > b) ? a : b ;

x = (x > 0) ? x : -x ;

int abs (int n)
   {
     return(n > 0 ? n : -n) ;
   }
```


Branchement multiple : switch - case

```
switch (expression)
   {
     case constante : instructions ;
                      ...
     case constante : instructions ;
                      ...
     ...
     default : instructions ;
               ...
   }
```

- **switch** compare successivement la valeur de l'expression aux constantes énumérées, exclusivement de type entier ou caractère
- **instruction** est une liste d'instructions et non un bloc
- dès qu'un point d'entrée est trouvé, toutes les instructions sont exécutées jusqu'à la fin ou jusqu'à l'instruction **break**

exemple :

```
switch (c)
   {
     case 'a' :  ajouter(a) ;
                 break ;
     case 's' :  supprimer() ;
                 break ;
     case 'q' :
     case 'Q' :  quitter() ;
                 break ;
     default :  menu() ;
   }
```


L'instruction **switch** se comporte comme un branchement calculé. Par conséquent, il ne faut *surtout pas oublier* de mettre les **break** aux endroits nécessaires :

```
switch (<expression>)
        {
          case <constante-1>:<suite d'instructions> break ;
          case <constante-2>:<suite d'instructions> break ;
          ...
          case <constante-n>:<suite d'instructions> break ;
          default :  <suite d'instructions>
        }
```

Si on ne met pas de **break**, l'exécution va continuer à la suite au lieu de sortir du **switch**, puisque les différentes constantes correspondent seulement à des étiquettes de branchement. Il y a parfois des cas où c'est l'effet souhaité ; mais il faut être prudent !

Exemples :


```
/* fonction qui teste si c est une voyelle */
int voyelle(char c)
   {
     int res ;

     switch (c)
         {
           case 'a' :
           case 'e' :
           case 'i' :
           case 'o' :
           case 'u' :
           case 'y' :
                       res = 1 ;
                       break ;
           default :
                      res = 0 ;
         }
     return(res) ;
   }
```

```
/* fonction de traitement d'une erreur */
void erreur(int x)
   {
   switch (c)
       {
         case 0 :
                  printf("erreur d'accès au fichier\n") ;
                  break ;
         case 1 :
                  printf("erreur de lecture\n") ;
                  break ;
         case 2 :
                  printf("problème à l'exécution\n") ;
                  break ;
         default :  ;
       }
   }
```

Itérations : while et do - while

```
while (expression)
   instruction ;
```

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

- **expression** a un résultat booléen
- **instruction** est soit une instruction simple (se terminant par ;), soit un bloc d'instructions (encadré par { et })
- **do - while** est identique à **while** sauf que le bloc d'instructions est exécuté **au moins 1 fois**
  ne pas oublier le point-virgule final de la ligne while

Exemples

while

```
int x ;
printf("valeur :  ") ;
scanf("%d",&x) ;
while (x > 0)
      {
        /* utilisation de la valeur */
        printf("valeur :  ") ;
        scanf("%d",&x) ;
      }
```

do - while

```
int x ;
do
   { printf("valeur :  ") ;
     scanf("%d",&x) ;
     /* utilisation de la valeur */
   }
while (x > 0) ;
```


Exemple :

```
/* conversion d'un entier en une suite de chiffres */
void conversion(int i, char s[])
   {
     int x = 0 ; /* index */
     int n = (sizeof(i) * 8) - 1 ;

     do {
          s[x++] = ((i < 0) ?  '1' :  '0') ;
          i <<= 1 ;
        }
     while (n--) ;

     s[x] = '\0' ;
   }
```

L'itération suivante est correcte :

```
char *pc ;
...
while ( *(pc++) ) ;
```

et recherche la fin d'une chaîne de caractères sur laquelle pointe la variable **pc**.

Itération : for

```
for (expr_init ; expr_test ; expr_incr)
   instruction
```

- **expr_init** : module d'initialisation, contient les instructions à exécuter **avant** le corps de l'itération.
- **expr_test** : expression évaluée **avant chaque** exécution du corps de l'itératon : quand **expr_test** est fausse, on sort de l'itération, pour continuer le traitement en séquence
  expr_test est une condition de **continuité** et non une condition d'arrêt.
- **expr_incr** : ce module contient les instructions exécutées **après** le corps de l'intération
- **instruction** est soit une instruction simple (se terminant par ;), soit un bloc d'instructions (encadré par { et })

exemple :

```
somme = 0 ;
for (i = 1 ; i <= n ; i++)
    { printf("valeur :") ; scanf("%f",&v) ;
      somme += v ;
    }
moy = (double) somme / (double) n ;
```

- si les deux modules expr_init et expr_incr comportent plusieurs instructions, celles-ci sont séparées par des virgules.
- de plus, les trois expressions figurant dans l'instruction for peuvent n'avoir aucun lien entre elles :

```
for (i = j = 0, k = 1, a = b ;
     (c = getchar()) != EOF ;
     i++, k <<= 1, p--)
     {
       ...
     }
```


Les deux premières lignes de l'exemple :

```
somme = 0 ;
for (i = 1 ; i <= n ; i++)
```

peuvent se regrouper sous la forme :

```
for (somme = 0, i = 1 ; i <= n ; i++)
```


L'opérateur virgule :

```
expression1, expression2, expression3, ...
```

- les expressions sont évaluées de la gauche vers la droite
- le résultat retourné est le résultat de l'évaluation de l'expression la plus à droite
- cet opérateur est surtout utile dans les structures de contrôle :

- itération **for**, par exemple le calcul de la valeur maximale des éléments d'un tableau de longueur tablen :

```
for (i = 1, max = tab[0], tmp = tab ; i < tablen ; tmp++, i++)
    if (max < *tmp) max = *tmp ;
```

- itération **while**, par exemple le calcul de la valeur maximale des éléments d'un tableau de longueur tablen :

```
max = tab[0], i = tablen, tab += tablen ;
while (tab--, i--)
      if (max < *tab) max = *tab ;
```



```
for (expr_init ; expr_test ; expr_incr)
      instruction
```

équivaut à :

```
expr_init ;
while (expr_test)
      {
        instruction
        expr_incr ;
      }
```

Les trois modules ci-dessous sont équivalents :

```
n = 1 ;
while (n <= 10)
   {
     printf("n=%d, n**2=%d\n",n,n*n) ;
     n++ ;
   }
```

```
n = 1 ;
do {
     printf("n=%d,n**2=%d\n",n,n*n) ;
     n++;
   }
while (n <= 10) ;
```

```
for (n = 1 ; n <= 10 ; n++)
    printf("n=%d,n**2=%d\n",n,n*n) ;
```


- Une ou plusieurs des trois expressions (**expr_init**, **expr_test**, **expr_incr**) peuvent être vides ; en particulier :

  `for ( ; ; )`

  est une boucle infinie !

- Initialisation d'un tableau par une itération **for** (voir aussi page 118) :

  `for (p = tableau ; p < tableau + TAILLE ; *(p++)=0) ;`

Question subsidiaire : que fait la séquence de code suivante :

`while (a += i, i++ < 10) ;` [1]

[1] La somme des 10 premiers entiers.

Programmer proprement avec les itérations

- le nombre d'itérations est calculable
  → instruction itérative simple
- le nombre d'itérations n'est pas calculable
  → instruction itérative conditionnelle

L'instruction break

- sortie d'une instruction **switch** (indispensable)
- sortie d'une itération **for** ou **while**

```
while (condition)
   {
     ...
     if (expr) break ;
     ...
   }
```

```
for ( ; condition ; )
   {
     ...
     if (expr) break ;
     ...
   }
```

Outil intéressant quand BIEN utilisé.

= porte ouverte à des programmes :

- mal écrits,
- incompréhensibles
- difficilement déboguables

  Merci de ne pas utiliser break dans une itération !

L'instruction continue

- provoque une nouvelle itération sans exécuter les instructions du reste du corps de l'itération.
- son utilisation est interdite pour les projets car le programme devient vite incompréhensible...

```
while (expr)
   {
     ...
     ...
     continue ;
     ...
   }
```


L'instruction goto

Attention aux bugs et aux programmes spaghetti !

L'instruction return

```
return ; return expression ; return(expression) ;
```

renvoie un résultat et termine l'exécution de la fonction

```
/* suppression de la parité */
char noparity (char c)
   { return(c & 0x7f) ;
   }
```

```
float max(float a, float b)
      { return( (a > b) ?  a :  b) ;
      }
```

```
ingt f(int a)
     { if (a > 10) return (10) ;
       ...
       return ...  ;
     }
```


`return` peut être utilisé dans une itération pour redonner directement la main à l'appelant.

Définition de la fonction *readline()*

```
int readline (char tampon[], int max)
   {
     int c, lg = 0
     while(1)
              {
                c = getchar() ;
                if (c == EOF) return(EOF) ;      /* fin de fichier */
                if (c == '\n') return(lg) ;      /* newline */
                if (lg >= max) return(max) ;     /* débordement */
                tampon[lg++] = c ;               /* cas normal */
              }
     return(1) ;
}
```

Appel de la fonction *readline()*

```
#define EOF -1
#define MAX = 256 /* taille du tampon */

char Buffer[MAX] ;

int main(void)

  {
    int size ;

    while ((size=readline(buffer,MAX)) != EOF)
          {
            ...
          }
  }
```

Introduction au langage C
└Les instructions

Lorsqu'il n'y a pas de concordance entre le type de la fonction et le type de la valeur retournée, il faut utiliser une conversion explicite (cast) :

```
double precision(int n)

  {
    ...
    return(          (n)) ;
  }
```

115 / 237

Voir **cast** page 62.

Voir les fonctions page 134.

Introduction au langage C
└Les instructions

L'instruction exit

```
exit(valeur) ;
```

- permet de terminer le programme en cours d'exécution, et de retourner un code d'erreur
- ce code d'erreur est récupéré par l'environnement appelant dans la variable ***status*** (prédéfinie)

```
int *p ;

p = (int *)(malloc((unsigned)(sizeof(int)*NB))) ;
if (p == NULL)
  {
    printf("problème d'allocation dynamique\n") ;
    exit(1) ;
  }
```

116 / 237

Voir la macro-fonction d'allocation de mémoire page 157.

10. Les structures de données (structures, unions)

Introduction au langage C
└Les structures de données (structures, unions)

Les structures de données (struct)

```
struct nomStructure
  { type listeVariable ;
    type listeVariable ;
    ...
  } ;
```

```
struct point
  { int x ;
    int y ;
  } ;
```

- aucune déclaration de variable
- aucune réservation d'espace mémoire
- juste une déclaration d'un type
- le nom du type est :

struct nomStructure

117 / 237

Les structures de données (struct)

```
struct nomStructure
   { type listeVariable ;
     type listeVariable ;
     ...
   } ;
```

```
struct point
   { int x ;
     int y ;
   } ;
```

- déclaration d'une variable de ce type :

```
struct point p1 ;
```

réservation d'espace mémoire pour la variable `p1` :

aucune initialisation par défaut (comme d'habitude)

Le constructeur `struct` permet de définir un type structure composé de plusieurs champs. Il est donc inoportun de souhaiter donner une valeur à ces champs lors de la définition de ce type.

Il est cependant possible d'initialiser les champs d'une variable de type structure lors de sa déclaration :

```
struct point p1 = {1, 10 } ;
```


Les structures (suite)

Définition d'un nom à cette structure :

```
typedef struct point
   { int x ;
     int y ;
   } POINT ;
```

Le nom du type est POINT.
Il est donc inutile de donner un nom à la structure :

```
typedef struct
   { int x ;
     int y ;
   } POINT ;
```


Les structures (suite)

Opérations autorisées entre variables de type structure :

- affectation :

```
p2 = p1 ;
```

recopie de toutes les valeurs des champs de `p1` dans les champs de `p2`.

- comparaison interdite !!

```
(p2 == p1)
```

- accès aux champs d'une variable de type structure :

```
nomVariable.nomChamp
```

```
p1.x
```


Il est tout à fait possible d'affecter deux variables de même type, cette affectation entraîne une copie physique (duplication).

Il est également possible d'écrire une déclaration incomplète de structure, c'est-à-dire dans laquelle ne figure pas la liste des champs. Par exemple :

```
struct personne ;
```

La principale utilisation de ce type de déclaration est la définition d'un pointeur :

```
struct personne *p ;
```

Une variable de ce type peut être utilisée dans une affectation, passée en paramètre à une fonction, ou retournée par une fonction. En revanche, elle ne peut pas être utilisée directement.

Les structures de données (union)

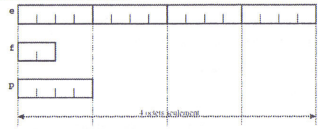
- Une union permet de représenter par un même type générique plusieurs types différents. On peut ainsi manipuler différents types de données dans une même zone mémoire.
- La liste des champs ne représente pas une concaténation d'objets mais différentes façons de considérer une même zone mémoire.

```
union {
        int e[4] ;
        short f ;
        char *p ;
      } uval ;
```



```
union {
        int e[4] ;
        short f ;
        char *p ;
      } uval ;
```

Un élément de ce type se représente en mémoire par une seule zone de 16 octets, cette zone est sémantiquement découpée en 3 champs :



- le champ `uval` est suffisamment grand pour contenir le type le plus long des 3 champs `e`, `f` et `*p`

- les champs d'une union doivent se partager leur espace mémoire
- on accède au champ d'une union par l'opérateur `.`
- un seul champ d'une union est "actif" à un moment donné
- les unions s'emploient le plus souvent à l'intérieur de structures et de tableaux

Exemple :

```
typedef struct place {
        enum tag {valeur,operateur,test} type ;
        union {
                int val ;
                char symb ;
                struct place *p ;
              } uval ;
} ELT ;
```

La déclaration :

```
ELT element ;
```

définit une variable `element` de type structure contenant deux champs : `type` et `uval`.

Exemple d'utilisation :

```
scanf("%s", rep) ;
switch(rep[0])
        {
          case '+' :
          case '-' :  element.type = operateur ;
                      element.uval.symb = rep[0] ;
                      break ;
          case '0' :
          case '1' :
                      element.type = valeur ;
                      element.uval.val = atoi(rep) ;
                      break ;
          ...
        }
```


Le champ **type** peut prendre ses valeurs dans l'énumération **tag** : **valeur**, **operateur** ou **test**.

En fonction de la valeur de ce champ, on aura accès soit au champ **val** (quand **type** contient **valeur**), soit au champ **sym** (quand **type** contient **operateur**), soit au champ **p** quand **type** contient **test**) du champ **uval** défini comme une union. Par exemple :

```
if (element.type == operateur)
   element.uval.sym = '+' ;
```

Le langage C ne vérifie pas l'utilisation appropriée d'un champ : on peut en fait accéder à n'importe quel champ de l'union quelle que soit la valeur du champ **type**. Faites attention aux incohérences du style :

```
if (element.type == operateur)
   element.uval.p = &i ;
```

Les opérateurs de manipulation d'adresses

opérateur	*valeur résiduelle*
`()`	`f()` la valeur retournée par la fonction d'adresse *f*
`[]`	`t[i]` l'élément d'indice *i* du tableau *t* (adresse $t + i$)
`.`	`struct.ch` la valeur du champ *ch* de la structure d'adresse *srtuct*
`->`	`p->ch` la valeur du champ *ch* de la structure pointée par *p*
`*`	`*p` l'objet pointé par *p*, "casté" dans le type de *p*
`&`	`&toto` l'adresse de l'objet *toto*

Les champs de bits

Il est possible de préciser, au bit près, la taille d'un champ de structure ou d'union, à condition cependant que ce champ soit de type `int` (signé ou non). Un tel champ est appelé **champ de bits**. Par exemple, la déclaration :

```
struct robot {
                    unsigned int actif : 1 ;
                    unsigned int numero : 3 ;
                    unsigned int position : 28 ;
}
```

définit une structure composée de 3 champs, de respectivement 1, 3 et 28 bits.

Si r est une variable déclarée par :

```
struct robot r ;
```

- le champ `r.actif`, codé sur 1 bit, peut prendre les valeurs 0 ou 1,
- le champ `r.actif`, codé sur 3 bits, peut prendre les valeurs de 0 à 7,
- le champ `r.position`, codé sur 28 bits, peut prendre les valeurs de 0 à $2^{28} - 1$.

La taille maximale d'un champ de bits est celle d'un entier, soit 16 ou 32 bits.

À l'aide

Les pages du manuel Unix sont divisées en plusieurs sections. Par exemple, sous Linux, on retrouve les sections suivantes :

1. commandes utilisateurs pouvant être exécutées quelque soit l'utilisateur
2. appels systèmes, c'est-à-dire les fonctions fournies par le noyau
3. fonctions des bibliothèques
4. périphériques, c'est-à-dire les fichiers spéciaux que l'on trouve dans le répertoire /dev
5. descriptions des formats de fichiers de configuration (comme par exemple /etc/passwd)
6. jeux
7. divers (macros, conventions particulières, ...)
8. outils d'administration exécutables uniquement par le super utilisateur (root)
9. autre section (spécifique à Linux) destinée à la documentation des services offerts par le noyau

Lorsque vous interrogez la documentation à propos d'un terme présent dans plusieurs sections (par exemple **passwd** qui est à la fois une commande et un fichier de configuration), si vous ne précisez pas le numéro de section, c'est toujours la section de numérotation la moins élevée qui sera affichée.

Chaque section possède une page d'introduction qui présente la section, disponible en tapant :

man <numeroSection> intro

Par exemple :

```
bash-3.00 $man 3 intro
INTRO(3)                Manuel du programmeur Linux                INTRO(3)

NOM
       intro - Introductions aux fonctions de bibliothèque.

DESCRIPTION
       Cette  section décrit toutes les fonctions de bibliothèque, à l'excep-
       tion des fonctions incluses dans la section 2, implémentant des appels
       systèmes.   Il  y  a  plusieurs groupes de fonctions, qui peuvent être
       identifiés par une lettre ajoutée au numéro de section :

       (3C)   Ces fonctions, celle de la section 2 et celles du  chapitre  3S
              sont  contenues dans la bibliotheque C libc, utilisée par cc(1)
              par défaut.

       (3S)   Ces fonctions correspondent à la bibliothèque stdio(3S).  Elles
              sont contenues dans la bibliothèque libc standard.

       (3M)   Ces  fonctions sont contenues dans la bibliothèque arithmétique
              libm, elles sont utilisées par défaut par le  compilateur  FOR-
              TRAN f77(1), mais pas par le compilateur C cc(1), qui nécessite
              l'option -lm.
...
```

Pour obtenir de l'aide sur la fonction ***printf()***, la commande :

```
man printf
```

ne donne pas accès à la fonction C , mais à la fonction ***printf()*** de Linux.

Il faut demander explicitement l'aide de la fonction ***printf()*** de C de la section 3 par :

```
man 3 printf
```