Retour sur l'anatomie des commandes du shell

Parmi les contributions majeures (contributions très admirées) des systèmes *NIX on peut mentionner l'abstraction des entrées/sorties des processus. En effet, les commandes écrivent (sur leurs sorties) et lisent (sur leur entrée) sans en connaître la véritable finalité, ce mécanisme permet deux choses :

- la **redirection**, c'est-à-dire la possibilité pour l'utilisateur final de choisir sur quoi ou depuis quoi les entrées/sorties s'effectuent rééllement
- la **composition**, c'est-à-dire la possibilité à partir de commandes élémentaires de composer celles-ci afin d'obtenir un fonctionnement plus complexe.

Si ces choses paraissent naturelles aujourd'hui, il faut se souvenir qu'à cette glorieuse époque on avait plutôt tendance à écrire des programmes complexes qui prenaient en chargent un maximum de fonctionnalités, c'est donc une révolution.

Flux standards

Definition (descripteurs)

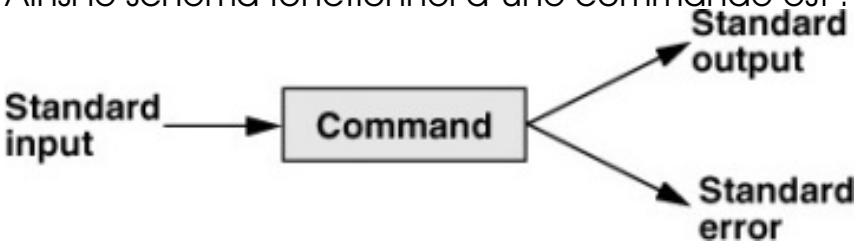
Tout processus (un processus est l'exécution d'un programme) a normalement la possibilité de lire ou d'écrire depuis ou sur des objets prédéfinis, les **flux standards** (*standard streams*) : leur **entrée standard** (*standard input*), leur **sortie standard** (*standard output*) et leur **sortie erreur standard** (*standard error*).

Le point important à noter ici est que les processus n'ont pas conscience de la réalité finale des entrées/sorties qu'ils réalisent à travers ces canaux. Ils lisent et écrivent sur ces **descripteurs** dont l'allocation n'est pas de leur fait.

Note

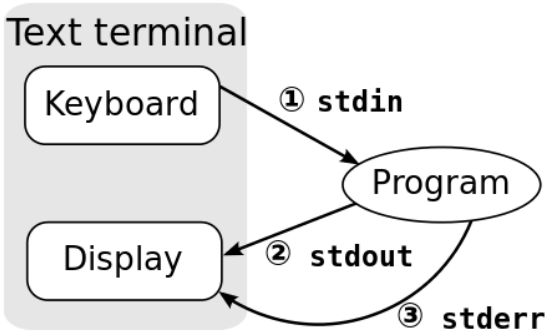
De ce fait une commande doit être vue comme un **filtre** qui lit des données en entrée et produit des données en sortie.

Ainsi le schéma fonctionnel d'une commande est :



source : informIT.com

Dans un fonctionnement ordinaire, les canaux sont connectés au clavier et à l'écran, c'est-à-dire les périphériques concernés par l'interaction :



source : wikipedia

Note

Rappelons que le clavier et l'écran sont considérés par le système comme des «fichiers».

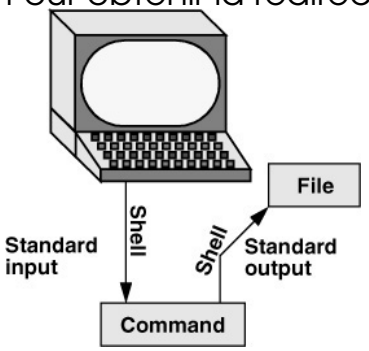
Redirections

Definition (Redirection)

La **redirection** est le procédé consistant à débrancher puis rebrancher un canal sur un «fichier» de sorte que la commande qui effectue ses opérations de lecture/écriture sur ce canal abstrait les réalise sur le «fichier» physique concerné.

Il est possible, entre autres *via* le shell de connecter, à son lancement, les flux d'une commande sur n'importe quel «fichier».

Pour obtenir la redirection de la sortie standard :



source : informIT.com

la syntaxe est :

commande **>** *fichier*

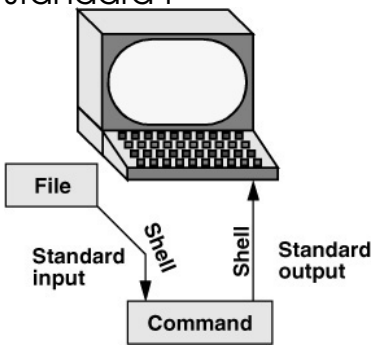
Exemple

```
$ echo bonjour
bonjour
$ echo bonjour > tst
$ cat tst
bonjour
$
```

Note

La commande **echo** permet d'obtenir en sortie l'affichage des ces arguments.
La commande **cat**, ainsi utilisée, permet d'obtenir en sortie le contenu d'un fichier passé en argument.

Pour obtenir, c'est-à-dire la redirection de l'entrée standard :



source : informIT.com

la syntaxe est :

commande **<** *fichier*

Exemple

```
$ cat words
bonjour
au revoir
aujourd'hui
hier
avant-hier
belle journée n'est-ce pas ?
$ grep jour
jour  # tapé par l'utilisateur
jour  # renvoyé par la commande
nuit  # tapé par l'utilisateur
$ grep jour < words
bonjour
aujourd'hui
belle journée n'est-ce pas ?
$
```

Note

La commande **grep**, ainsi utilisée, permet d'obtenir en sortie le contenu filtré de son entrée standard. Le filtre qui opère ici est la recherche dans chaque ligne de texte de la présence de l'argument (la chaîne `jour`) qui provoque alors la sortie de la ligne.

Bien entendu on peut «mixer» les deux.

Exemple

```
$ grep jour < words > results
$ cat results
bonjour
aujourd'hui
belle journée n'est-ce pas ?
$
```


Attention

Il ne faut pas se méprendre, si les deux commandes **grep** `jour < words` et **grep** `jour words` produisent le même résultat, leur fonctionnement n'est pas le même!
Dans le premier cas, la commande n'a que deux arguments (`grep` et `jour`), lit son entrée standard et écrit sur sa sortie standard.
Dans le second cas la commande a trois arguments (`grep`, `jour` et `words`), ouvre le fichier donné en troisième argument, filtre son contenu et écrit le résultat sur sa sortie standard.
Dans le premier cas le shell effectue une redirection alors qu'il n'y en a pas dans le second cas.

Il est aussi possible de rediriger la sortie erreur *via* :

commande **2>** *fichier*

Attention

Si les messages standards et les messages d'erreurs sont par défaut affichés à l'écran, il ne faut pas se méprendre car il le sont *via* des canaux différents (lesquels sont connectés au même dispositif).

```
$ ls blabla toto
ls: blabla: No such file or directory # erreur
toto # message «normal»
$
```


Exemple

```
$ ls blabla toto
ls: blabla: No such file or directory
toto
$ ls blabla toto 2> error
toto
$ cat error
ls: blabla: No such file or directory
$ ls blabla toto > result
ls: blabla: No such file or directory
$ cat result
toto
$ ls blabla toto > result 2> error
$ cat error
ls: blabla: No such file or directory
$ cat result
toto
$
```

Il existe d'autres mécanismes de redirection, vous les étudierez (peut-être) lors des travaux pratiques.
L'essentiel du mécanisme est ici décrit.

Composition : les tubes

L'autre mécanisme offert par les redirections est la composition de commandes.
Par exemple, si je souhaite obtenir le nombre de lignes contenant une certaine chaîne de caractères je peux toujours faire :

```
$ grep jour words > results
$ wc -l < results
       3
$ rm results
$
```

Un inconvénient majeur est d'avoir à passer par un fichier intermédiaire (`results`), lequel peut-être volumineux ou entrer en conflit avec un fichier déjà existant.

La solution est d'établir un canal de communication direct entre les deux commandes *via* un tube.

Definition (tube (*pipe*))

Object de communication unidirectionnel assurant le transport de données de bout en bout et la synchronisation des processus concurrents réalisant les lectures et les écritures.

La syntaxe suivante assure l'établissement d'un tube anonyme (*anonymous pipe*) entre la sortie standard de la première commande et l'entrée standard de la seconde :

$commande_1$ | $commande_2$

Ainsi, la fonctionnalité recherchée peut-être réalisée par la simple commande suivante :

```
$ grep jour words | wc -l
       3
$
```

Note

On notera que la création et la suppression du tube sont prises en charge de façon transparente par le shell, aucune des deux commandes n'effectue quelque chose de particulier.

Les scripts shell

On a déjà mentionné qu'un shell exécute des commandes, soit interactivement depuis un terminal soit par interprétation d'un script.

Definition (script)

Un script shell est un fichier texte contenant des commandes du shell. Il existe deux types de commandes : des commandes externes (vous en avez déjà connaissance) et les commandes internes (c'est-à-dire que leur code fait partie du shell lui-même).

Exemple

```
#!/bin/sh
echo mon premier script
```


Exemple

```
#!/bin/sh
echo mon premier script
```

Le caractère `#` est un début de commentaire, lequel débute en ce caractère et se termine en fin de ligne. Lorsque la première ligne est un commentaire qui commence par `#!`, ce commentaire permet de spécifier le shell qui servira à interpréter les commandes du script. Nous avons choisi d'utiliser le shell standard `sh`, l'autre option commune est le `bash` (une extension du shell standard).

Afin que ce script soit exécutable directement, il faut modifier les droits d'accès au fichier correspondant. Par exemple :

Exemple

```
$ chmod u+x script1.sh
$
```

Cette commande permet de rajouter pour le propriétaire du fichier (**u**ser) le droit d'exécution (e**x**ecution). Il n'est pas strictement nécessaire de rendre le fichier exécutable, car il peut être interprété par différents biais mais nous ne creuserons pas plus loin ce point.

Une fois rendu exécutable son exécution produit:

```
$ ./script1.sh
mon premier script
$
```

La commande `echo` comme on peut l'observer affiche sur sa sortie ses arguments.

Exemple

```
#!/bin/sh
echo "Voici la liste des fichiers"
ls
echo "----fin de liste----"
```

dont l'exécution produit :

```
$ ./monls
Voici la liste des fichiers
es.h            def             incbad.c
...
----fin de liste----
$
```


Question

Dans le second script nous avons utilisé " afin de délimiter la chaîne à afficher, pourquoi ?

Bien qu'il n'y ait pas d'effet observable, la différence est importante car dans le premier script la commande echo reçoit plusieurs arguments (on rappelle que les arguments sont les mots séparés par des caractères d'espacement, alors que dans le second script la commande echo ne reçoit qu'un seul argument (la chaîne dans son entier, y compris les espaces).

Exemple

```
#!/bin/sh
echo un message
echo un    autre         message
echo "un␣␣␣autre␣␣␣␣␣␣␣␣message"
```

produit :

```
$ chmod u+x script2.sh
$ ./script2.sh
un message
un autre message
un   autre      message
$
```


La seconde commande affiche le message `un autre message` quand bien même la ligne correspondante fait apparaître un nombre quelconque de caractères d'espacement. Comme rien n'est spécifié de particulier, chaque mot est considéré comme un argument, et

Definition (`echo`)

La commande `echo` a pour effet d'afficher l'ensemble de ses arguments en les séparant par un simple espace.

Definition (")

Le caractère " permet d'ouvrir et fermer la description d'une chaine de caractère qui sera lue par le shell comme un seul argument.

On verra qu'il existe d'autres possibilités.

Arguments d'un script

Les arguments passés à l'exécution d'un script lui sont accessible *via* des variables particulières :

Exemple

```
#!/bin/sh
echo "Le nombre d'arguments est $#"
echo "La liste des arguments est : $*"
echo "Le premier est $0"
echo "Le second est $1"
echo "Le troisième est $2"
```

Les variables du shell sont lues en spécifiant leur nom précédé du signe `$`.

Son exécution produit :

```
$ ./arguments a "b c" d
Le nombre d'arguments est 3
La liste des arguments est : a b c d
Le premier est ./arguments
Le second est a
Le troisième est b c
$
```

On peut observer que le premier argument est la commande elle-même et a pour indice 0.
L'autre remarque est que les variables sont lues, on dit **expansées** (*expanded*), y compris à l'intérieur d'une chaîne délimitée par `"`.

Expansion des variables

Definition (Expansion)

On appelle **expansion des variables** l'un des processus réalisé par le shell sur le texte que constitue la ligne de commande. Elle consiste en le remplacement des variables par leur valeur.

Si " permet d'empêcher le découpage en mots par le shell, le caractère ' empêche en plus l'expansion des variables.

Exemple

```
#!/bin/sh
VAR=toto
echo --   $VAR   --
echo "--␣␣␣$VAR␣␣␣--"
echo '--   $VAR   --'
```

produit:

```
$ expansion
-- toto --
--   toto   --
--   $VAR   --
$
```

Terminaison et statut

Sous *NIX, tout processus (on rappelle qu'il s'agit de l'exécution d'un programme) qui se termine indique, d'une manière ou d'une autre, la raison de sa terminaison.
Il peut y avoir différentes raisons pour une terminaison :

- terminaison normale, c'est-à-dire que l'exécution a été conduite jusqu'au bout,
- terminaison brutale, lorsque le processus a été interrompu par un évènement particulier (on verra plus tard).

Il est donc important que l'on puisse déterminer comment un processus s'est-il terminé. C'est le rôle du **code de sortie** ou **statut** (*exit status*).

Definition (code de sortie)

Le code de sortie d'une commande est une valeur indiquant la qualité de la terminaison d'un processus.

Il s'agit d'une sorte de valeur de retour un peu à la manière dont les fonctions retournent des valeurs dans les programmes.
Ici c'est le programme qui renvoie une valeur à son environnement (dont la sémantique est fixée : quel type de comportement le programme a t-il eu lors de son exécution ?).

⚠ Attention

Le concept de «terminaison normale» prête souvent à confusion. Le programme peut avoir rencontré une erreur et cela peut-être une terminaison normale. Par exemple, il est normal que `ls` ne puisse pas, par exemple, afficher les informations d'un «fichier inexistant», ou alors que `grep` ne trouve pas une suite de caractères dans un fichier.
Ces comportements sont normaux car ils sont «prévus» dans le code des programmes correspondants.
Ce qui est anormal c'est d'interrompre l'exécution par un moyen ou par un autre (vous connaissez peut-être `^-C` ou la commande `kill` ? Ceci provoque habituellement l'arrêt immédiat et sans condition du processus. C'est un comportement «anormal», «imprévu»...

Definition (terminaison normale)

La terminaison normale d'un script peut-être indiqué *via* la commande (interne) :

`exit` *valeur*

où *valeur* peut prendre n'importe quelle valeur entre 0 et 127. Cette valeur peut-être interprétée comme un niveau d'erreur, où 0 signifie tout s'est déroulé sans problème et $\neq 0$ que la fonctionnalité n'a pu être réalisée car les conditions de sa réalisation n'ont pu être réunies.
Si un script ne précise pas de code de sortie, c'est celui de la dernière commande exécutée qui sera renvoyée.
La valeur du code de sortie de la dernière commande exécutée est disponible dans la variable `$?`.

Exemple

```
$ ls toto
toto
$ echo $?
0
$ ls blabla
ls: blabla: No such file or directory
$ echo $?
1
$ echo $?
0
$
```


Quelques commandes internes du shell

Le shell fournit un véritable langage de programmation. Les scripts que nous avons jusqu'à présent réalisés sont très primitifs et n'enchaînaient qu'une bête succession de commandes. Il est possible, comme dans tous les langages de programmation, d'utiliser des structures de contrôle de l'exécution (alternatives, sélections, fonctions, boucles).

Attention

Comme tout langage de programmation, celui du shell est plus particulièrement dédié à certaines tâches. Bien qu'on puisse en théorie effectuer n'importe quel calcul, le défi serait immense.
Le langage du shell est dédié au contrôle de l'enchaînement de commandes; il n'est pas adapté au calcul numérique (même si on peut en faire si l'on souhaite).

Nous n'étudierons ici pas toutes les constructions, ceci devrait faire l'objet de travaux pratiques. Il s'agit de comprendre les mécanismes sous-jacents.

`if` en shell

L'aternative peut prendre plusieurs formes, dont :

if $commande_{test}$
then
  $commande_{si=0}$
else
  $commande_{si\neq 0}$
fi

Note

Il est très important de noter que l'argument du `if` est une commande! C'est son code de sortie qui servira à effectuer le bon branchement. Si celui-ci est égal à 0, c'est la partie `then` qui sera exécutée, sinon `else`.
On remarquera aussi que le `if` est clos par un `fi`.

Exemple

```
#!/bin/sh
if ls "$1"
then
  echo "yes"
else
  echo "no"
fi
```

dont l'exécution donne :

```
$ ./myif.sh toto
toto
yes
$ ./myif.sh blabla
ls: blabla: No such file or directory
no
$
```


Si l'on considère que les sorties de `ls` polluent l'exécution, on peut avantageusement utiliser le puits sans fond `/dev/null` :

Exemple

```
#!/bin/sh
if ls "$1" > /dev/null 2> /dev/null
then
  echo "yes"
else
  echo "no"
fi
```

dont l'exécution donne :

```
$ ./myifdevnull.sh toto
yes
$ ./myifdevnull.sh blabla
no
$
```

les tests en shell

Pour effectuer des tests plus «standards» il existe une commande dédiée, `test` :

```
$ test 0 = 0
$ echo $?
0
$ test 0 = 1
$ echo $?
1
$
```


Exemple

```
#!/bin/sh
if test $1 = toto
then
  echo "trouvé"
else
  echo "pas trouvé, essaie encore!"
fi
```

```
$ ./cherche.sh tata
pas trouvé, essaie encore!
$ ./cherche.sh toto
trouvé
$
```

Injection de code

Exemple

```
#!/bin/sh
if test $1 = toto
then
  echo "trouvé"
else
  echo "pas trouvé, essaie encore!"
fi
```

Question

Ce code comporte un problème (sérieux), sauriez-vous trouver lequel ?

On peut l'appeler de sorte à «injecter» du code à l'intérieur du script...

```
$ cherche.sh "a = a -o a "
trouvé
$
```

Ce qui n'était pas l'intention première! L'expansion des variables est délicate! L'argument `-o` permet d'obtenir le «ou».
Une correction serait :

```
if test "$1" = toto
```

la commande [

La commande `test` est disponible sous une autre forme (un peu plus courante d'usage) :

```
[ test ]
```

et oui il existe bien une commande de nom `[`.

```
$ ls /bin/[
/bin/[
$ [ a = b ]
$ echo $?
1
$ [ a = a ]
$ echo $?
0
$
```


Retour vers le langage

Les variables

En C, une variable doit être déclarée ou définie avant d'être utilisée.

Forme d'une définition

type identificateur;

comme par exemple :

Exemple

```
int nombre_eleves;
```


Types

Parmi les types de base du C on trouve :

- **char**,
- **short**,
- **int**,
- **long**,
- **float**,
- **double**

Note

L'étude plus systématique des types du C est reportée à plus tard.

Identificateur

Definition (identificateur)

- séquence de caractères alphanumériques,
- et de caractères _,
- mais qui ne peut commencer par un chiffre.

⚠ Attention

Il est légal d'utiliser _ en tête d'un identificateur mais cet usage n'est pas recommandé.
Le langage C définit des mots-clés et réservés. Aucun identificateur ne doit correspondre à un mot-clé; il faut éviter les mots réservés.

Exemple

Sont valides :

```
int the_number_of_the_beast;
double pi;
```

Sont invalides :

```
int 666_is_the_beast;
short else;
```

Ne sont pas recommandés :

```
int errno;
int __i_am_the_beast;
int MAX;
```

Initialisation

Definition (Syntaxe «simple» d'initialisation)

type identificateur = valeur;

Exemple

```
int note_minimale = 0;
double presque_pi = 3.1415926;
```

Ces exemples utilisent des **littéraux**, c'est-à-dire des valeurs **constantes** explicitement exprimées dans le code.

Le langage ne requiert pas l'initialisation explicite des variables. Certaines variables sont implicitement initialisées...

⚠ Attention

Le contenu des variables automatiques (voir p.228) non initialisées est non déterminé. Prenez la peine d'initialiser toutes les variables...

Valeurs

En C une **variable** désigne un **contenant** qui possède une **valeur** (qui peut être modifiée si le contenu est modifiable...).
Il n'y a pas de référence en C, une variable désigne :

- son contenu, si l'identificateur est une **r-value**,
- son contenant, si l'identificateur est une **l-value**.

Une **variable est une l-value** si elle est en position d'être modifiée. l-value = left-value, historiquement lorsqu'elle est placée à gauche dans une affectation.
Une **variable est une r-value** si elle est en position d'être lue. Par symétrie avec l-value.

Expression

Definition (Expression)

Séquence bien formée d'**opérateurs** et d'**opérandes** et qui spécifie le calcul d'une **valeur** typée.

Exemple

```
13
i
a + b
2*PI*radius
4./3.*PI/pow(radius,3)
(3+a)*x
a = b // Une affectation est une expression!
```

Toutes les expressions (à de très rares près) ont une valeur.

Les opérateurs

Le langage C définit de nombreux opérateurs afin construire des expressions.

Opérateurs postfixes

Ceux-ci sont :

- `[]` (voir p. 362),
- `()` (voir p. **??**),
- `.` (voir p. 388),
- `->` (voir p. **??**),
- `++`,
- `--`
- et `{}` (voir p. **??** et p. **??**).

Opérateurs unaires postfixes ++ et --

Ces opérateurs sont postfixes car ils apparaissent à la suite d'une expression.

Definition (++, --)

L'opérateur postfixe ++ dit de post-incrémentation (respectivement -- dit de post-décrémentation) appliqué à une variable modifiable sur laquelle une arithmétique est définie construit une expression dont la valeur est celle de la variable et à pour effet secondaire d'ajouter (respectivement d'enlever) 1 au contenu de la variable (attention à l'arithmétique des pointeurs, voir p.**??**).

Exemple

```
#include <stdio.h>

int main(void) {
  int i = 0;
  printf("%d\n",i++);
  printf("%d\n",i--);
  printf("%d\n",i);
}
```

produit :

```
$ ./postfix
0
1
0
```

⚠ Attention

L'ordre d'évaluation des opérandes de la plupart des opérateurs n'est pas défini par conséquent il ne faut pas utiliser des opérateurs à effet secondaire sur la même variable dans une même expression; le comportement est indéfini.

Exemple à ne pas suivre

```
a = i++ + i++*2;
```


Opérateurs unaires préfixes

Ce sont les opérateurs sont préfixes car ils apparaissent avant l'expression sur laquelle ils s'appliquent :

- `++`,
- `--`,
- `sizeof`, voir p. **??**,
- `_Alignof`, voir p. 422,
- `&`, voir p. 262,
- `*`, voir p. 262,
- `+`,
- `-`,
- `~`
- et `!`.

Opérateurs préfixes ++ et --

Definition (++, --)

Si l'opérande est une variable modifiable d'un type sur lequel est défini une arithmétique, alors l'opérateur ++ dit de préincrémentation (respectivement -- dit de prédécrémentation) ajoute (respectivement enlève) 1 à la variable (attention à l'arithmétique des pointeurs, voir p. 288) et la valeur de l'expression est celle de la variable après opération. Ces opérateurs ont donc un effet secondaire.

⚠ Attention

La remarque pour leurs cousins postfixes s'applique aussi.

Exemple

```
#include <stdio.h>

int main(void) {
  int i = 0;
  printf("%d\n",++i);
  printf("%d\n",--i);
  printf("%d\n",i);
}
```

produit :

```
$ ./prefix
1
0
0
```

Opérateurs unaires + et −

Definition (+, −)

Le résultat de l'opérateur unaire + (respectivement −) est la valeur (respectivement de l'inverse arithmétique) de son opérande (après promotion).

Exemple

```
#include <stdio.h>

int main(void) {
  int i = 12;
  printf("%d\n",+i);
  printf("%d\n",-i);
  i = -12;
  printf("%d\n",+i);
  printf("%d\n",-i);
}
```

produit :

```
$ ./unary
12
-12
-12
12
$
```

Opérateur unaire ~

Definition (~)

L'opérateur préfixe unaire ~ dit opérateur de complémentation bit-à-bit a pour effet d'inverser indépendamment chaque bit de la représentation de son opérande, laquelle doit impérativement être d'un type entier.

Exemple

```
#include <stdio.h>

int main(void) {
  int i = 12;
  int j = ~i;
  printf("%08x␣%08x\n",i,j);
}
```

produit :

```
$ ./compbit
0000000c fffffff3
$
```

Opérateur !

Definition (!)

L'opérateur préfixe unaire `!` dit de négation logique produit :

- la valeur 0 de type `int`, si la valeur de son opérande (de type scalaire) n'est pas égale à 0;
- le valeur 1 de type `int`, si la valeur de son opérande (de type scalaire) est égale à 0.

Opérateurs `sizeof` et `_Alignof`

Definition (`sizeof`)

L'opérateur préfixe unaire `sizeof` s'applique à toute expression ou type, à l'exception des types des fonctions, des types incomplets et des champs de bits. Son résultat est un type entier défini comme `size_t` (en-tête `<stddef.h>`) et la valeur de l'expression est la taille exprimée en octets du type de l'opérande ou du type.
Lorsque l'opérande est un type, celui-ci doît être spécifié entre parenthèses.
De plus `sizeof(char)`, `sizeof(unsigned char)` ou `sizeof(signed char)` ainsi que tous les dérivés qualifiés fournit la valeur 1.
Pour les tableaux, la taille est celle du tableau (voir p. 367 et p. **??**).

Exemple

```
#include <stdio.h>

int main(void) {
  int i = 12;
  double d = 0.0;
  printf("%lu\n",sizeof i);
  printf("%lu\n",sizeof d);
  printf("%lu\n",sizeof(123*4U/2.5));
  printf("%lu\n",sizeof(long long));
}
```

produit :

```
$ ./sizeof
4
8
8
8
$
```


⚠ Attention

Les valeurs produites par cet opérateur sont dépendantes de l'implémentation (sauf pour les types `char`).

⚠ Attention

L'opérateur `sizeof` **n'évalue pas l'opérande**, il n'en évalue que le type. Aucun code n'est émis à la compilation... Sauf bien sûr pour les types tableaux de longueur variable... (voir p.375)

L'opérateur `_Alignof`

Definition (`_Alignof`)

Cet opérateur renvoie la contrainte d'alignement du type qui le suit. Le type ne peut être incomplet ni celui d'une fonction.
La valeur renvoyée est du type entier `size_t` (en-tête `<stddef.h>`).
Pour les types tableaux, la contrainte est celle de ses éléments.

Pour plus de détails en ce qui concerne l'alignement, se reporter à p. 422.

Exemple

```
#include <stdio.h>

int main(void) {
  printf("%lu\n",_Alignof(long long));
  printf("%lu\n",_Alignof(char));
  printf("%lu\n",_Alignof(int));
  printf("%lu\n",_Alignof(int *));
}
```

produit :

```
$ ./align
8
1
4
8
$
```

Les opérateurs multiplicatifs ⋆, / et %

Definition (⋆, /, %)

Il s'agit d'opérateurs binaires infixes et leurs opérandes doivent être d'un type arithmétique. Pour % la contrainte de type est que les opérandes doivent avoir un type entier.
⋆ construit le produit arithmétique.
/ construit la division «naturelle» pour les flottants. Dans le cas des entiers, le résultat est celui de la division «naturelle» à laquelle la partie fractionnelle est éliminée (cette opération est parfois appelée «arrondi par troncature»). Son second argument ne doit pas être nul, sinon le comportement est non défini.
% construit le reste de la division entière, la valeur produite est positive. Son second argument ne doit pas être nul, sinon le comportement est non défini.

Exemple

```
#include <stdio.h>

int main(void) {
  double f1 = 5;
  double f2 = 3;
  int i = 5;
  int j = 3;
  printf("%f␣%f\n",f1*f2,f1/f2);
  printf("%d␣%d␣%d\n",i*j,i/j,i%j);
  f1 = -5;
  f2 = 3;
  i = 5;
  j = -3;
  printf("%f␣%f\n",f1*f2,f1/f2);
  printf("%d␣%d␣%d\n",i*j,i/j,i%j);
}
```

produit :

```
$ ./mult
15.000000 1.666667
15 1 2
-15.000000 -1.666667
-15 -1 2
$
```

Attention

La division des flottants peut surprendre. Il ne faut pas oublier qu'il s'agit d'approximations de rééls et que les calculs sont effectués avec une certaine précision. Pour les détails, se rapporter au cours de première année (PF1).

Les opérateurs additifs +, –

Definition (+, –)

L'opérateur + dit d'addition (respectivement – dit de soustraction) calcule, pour des opérandes de type arithmétique, l'addition (respectivement la soustraction) des deux opérandes.
L'opérateur + fonctionne aussi lorsque l'une des deux opérandes est un type pointeur, voir p. 288.
L'opérateur – fonctionne aussi lorsque les deux opérandes sont de type pointeur ou la première est de type pointeur et la seconde de type entier, voir p.288.

⚠ Attention

L'arithmétique des entiers contient un piège important. Si l'arithmétique des entiers non signés est l'arithmétique modulo, pour les entiers signés tout dépassement produit un comportement non défini. Écrire un code qui prend en compte cette particularité nécessite une certaine expertise, d'autre part il est fréquent que l'arithmétique implémentée est celle du complément à 2, c'est pourquoi elle est souvent ignorée. Mais il ne faut pas l'ignorer!

Autrement dit :

```
int i = INT_MAX;
i++;
```

produit un comportement non défini!

Les décalages bit-à-bit << et >>

Definition (<< et >>)

Ces opérateurs binaires infixes n'opèrent que sur les types entiers.
Si la valeur de du second argument est négative ou plus grande que le nombre de bits de la représentation du type de la première opérande, le comportement est non défini.
Pour << la suite de bits est poussée vers la «gauche» en faisant rentrer des 0s, de sorte que si l'opérande gauche vaut n et que l'opérande droite vaut d, la valeur produite est égale à $n.2^d$. La valeur produite est réduite modulo si l'opérande gauche est non signée.
Si l'opérande gauche est signée et que l'opération produit un dépassement, alors le comportement est non défini.

Definition

Pour >>, dans le cas où l'opérande gauche est de type non signée ou signée mais de valeur positive ou nulle, la suite de bits est poussée vers la «droite» en laissant rentrer des 0s, de sorte que la valeur produite soit la partie entière du quotient $n/2^d$. Si l'opérande gauche est signée et de valeur négative, le comportement est défini par l'implémentation.

Exemple

```
#include <stdio.h>

int main(void) {
  unsigned short us = 1;
  short s = 1;
  for (int i=0; i<8*sizeof us-1; i++) {
    printf("%u␣",us);
    us <<= 1;
  }
  for (int i=0; i<8*sizeof us; i++) {
    printf("%u␣",us);
    us = us >> 1;
  }
  putchar('\n');
  for (int i=0; i<8*sizeof s-1; i++,s = s<<1) {
    printf("%d␣",s);
  }
  for (int i=0; i<8*sizeof s; i++,s = s>>1) {
    printf("%d␣",s);
  }
  putchar('\n');
}
```

produit :

```
$ ./decal
1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768
     16384 8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1
1 2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 -3
    2768 -16384 -8192 -4096 -2048 -1024 -512 -256 -128 -64
    -32 -16 -8 -4 -2 -1
$
```


Les opérateurs relationnels >, >=, <, <=

Definition (>, >=, <, <=)

La valeur produite par ces comparaisons est de type `int` et vaut 0 si le test est faux et 1 sinon.
Dans le cas des nombres, la relation testée est triviale. Pour les pointeurs, le résultat dépend de la localisation relative des adresses, cela vaut pour les pointeurs vers des éléments d'un même tableau, l'ordre est celui des indices des éléments respectifs; même chose pour les pointeurs sur des membres de structure et pointeurs sur unions. Dasn les autres cas, le comportement est non défini.

Les opérateurs d'égalité ==, !=

Definition (==, !=)

La valeur produite par ces comparaisons est de type `int` et vaut 0 si le test est faux et 1 sinon.
Dans le cas des nombres, la relation testée est triviale.
Pour les pointeurs aussi, à condition que les types pointés soient les mêmes (aux qualifications près), ou que l'un des pointeurs soit un `void *`.

L'opérateur de conjonction bit-à-bit &

Definition (&)

Cet opérateur binaire infixe nécessite des opérandes de type entier et produit la valeur correspondant au et bit-à-bit des deux opérandes.

Cet opérateur est souvent employé pour tester la présence d'un bit dans un mot.

Exemple

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

#define FLAG_READY (0x1)

int main(void) {
  srandom(1);
  for (int i=0; i<5; i++) {
    if (! (random()&FLAG_READY) ) puts("not ready");
    else                          puts("ready");
  }
}
```


produit :

```
$ ./et
ready
not ready
ready
ready
ready
$
```

L'opérateur de disjonction exclusive bit-à-bit ^

Definition (^)

Cet opérateur binaire infixe nécessite des opérandes de type entier et produit la valeur correspondant au ou-exclusif bit-à-bit des deux opérandes.

Cet opérateur est fréquemment employé pour inverser des bits dans des masques de bits.

Exemple

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

#define ONOFF 1

void affiche(int m) {
  if (m&1) printf("on␣");
  else     printf("off␣");
}

int main(void) {
  int m = 0;
  for (int i=0; i<5; i++) {
    affiche(m);
    m = m^ONOFF;
  }
  putchar('\n');
}
```

produit :

```
$ ./ouex
off on off on off
$
```


L'opérateur de disjonction bit-à-bit |

Definition (|)

Cet opérateur binaire infixe nécessite des opérandes de type entier et produit la valeur correspondant au ou bit-à-bit des deux opérandes.

Cet opérateur est fréquemment employé pour fabriquer des masques de bits.

Exemple

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

#define READ_MODE (1U<<0)
#define WRITE_MODE (1U<<1)

void openbidule(unsigned int mode) {
  printf("ouverture en mode : ");
  if (mode&READ_MODE)  printf("read ");
  if (mode&WRITE_MODE) printf("write ");
  putchar('\n');
}

int main(void) {
  openbidule(READ_MODE);
  openbidule(WRITE_MODE);
  openbidule(READ_MODE | WRITE_MODE);
}
```


produit :

```
$ ./ou
ouverture en mode : read
ouverture en mode : write
ouverture en mode : read write
$
```

Et logique

Definition (&&)

L'opérateur binaire de conjonction logique && permet d'obtenir la valeur entière 1 si ces deux opérandes (scalaires) sont différentes de 0 et 0 sinon.
Cet opérateur garantit que les opérandes sont évaluées de gauche à droite et s'arrête dès que le résultat peut être déterminé (évaluation paresseuse).

Exemple

```
#include <stdio.h>

int main(void) {
  if (printf("bonjour ") && printf("le monde\n"))
    printf("ok\n");
  else
    printf("pas ok");
}
```

```
$ ./logicaland
bonjour le monde
ok
$
```

Sachant que `printf` renvoie le nombre de caractère envoyés en sortie...

Ou logique

Definition (||)

L'opérateur binaire de conjonction logique || permet d'obtenir la valeur entière 1 si l'une des deux opérandes (scalaires) sont différentes de 0 et 0 sinon. Cet opérateur garantit que les opérandes sont évaluées de gauche à droite et s'arrête dès que le résultat peut être déterminé (évaluation paresseuse).

Je ne veux pas travailler...
— Pink Martini —

Les opérateurs logiques produisent une évaluation paresseuse.

Definition (Évaluation paresseuse (*lazy evaluation*))

Technique d'implantation consistant à n'évaluer les arguments que lorsque leur valeur est strictement nécessaire.
Elle porte aussi le nom d'évaluation retardée ou par nécessité.

`a && b` est «fausse» dès que `a` est «fausse», il est alors inutile d'évaluer `b`.
`a || b` est «vraie» dès que `a` est «vraie», il est alors inutile d'évaluer `b`.

Exemple

```
#include <stdio.h>

int main(void) {
  if (printf("") && printf("666\n"))
    printf("ok\n");
  else
    printf("pas␣ok\n");
}
```

```
$ ./logicaland2
pas ok
$
```


Question

```
int x; int y;
...
if ( (x>5 && y<10) || (x<7) || (y%2==0) )
   puts("ok");
```

- Donnez un exemple de valeurs qui produisent l'affichage `ok` après l'évaluation de la 1ère (puis 2nde, puis 3ième) sous-expression.
- Quels sont les intervalles qui conduisent à l'affichage après évaluation de chaque sous-expression ?

Opérateur conditionnel

Definition (?:)

L'opérateur (ternaire) `?:` construit une expression dont la valeur une celle d'une de deux expressions dont le choix est résultat de l'évaluation d'une condition. La syntaxe est :

condition `?` $expression_{true}$ `:` $expression_{false}$

Exemple

```
a = x==y ? b : c;
```

qui affecte `b` à `a` si `x` est égale à `y` et `c` sinon.

Cet opérateur permet d'écrire de façon plus concise (et lisible lorsqu'on en a l'habitude et que l'on ne pratique pas d'abus) certaines alternatives comme :

Exemple

```
if (flag)
  x = 12;
else
  x = 27*y;
```

s'écrit plus concisément :

```
x = flag ? 12 : 27*y;
```

On peut aussi l'utiliser dans les cas suivant :

```
if (flag)
  printf("ici\n");
else
  printf("la\n");
```

qui s'écrit aussi :

```
flag ? printf("ici\n") : printf("la\n");
```

ou encore :

```
if (flag)
  printf("je␣vais␣ici\n");
else
  printf("je␣vais␣la\n");
```

```
printf("je␣vais␣%s\n",flag?"ici:"la");
```


Opérateurs d'affectation

Definition (Affectation)

On appelle **affectation** (*assignment*) l'opération consistant à modifier le contenu d'une variable. Le C possède plusieurs opérateurs d'affectation : `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=` et `|=`.

Note

Ces opérateurs construisent des **expressions**... D'autre part, ils ont une associativité droite.

Associativité

L'associativité des opérateurs la propriété d'ordre d'évaluation desdits opérateurs lorsqu'une expression contient plusieurs d'entre eux. Par exemple, comment est évaluée `a+b+c` ? Ou `a = b = c` ?
L'associativité gauche signifie qu'une expression comme `a + b c` s'évalue comme `((a + b) + c)`.
L'associativité droite signifie qu'une expression comme `a = b = c` s'évalue comme `(a = (b = c))`.

Affectation simple

L'opérateur correspondant est =, il permet la simple modification du contenu d'une variable.

Exemple

```
#include <stdio.h>

int main(void) {
  int a = 5;
  int b = 7;
  int c = 9;
  printf("%d␣%d␣%d\n",a,b,c);
  a = 10;
  printf("%d␣%d␣%d\n",a,b,c);
  a = b = c = 4;
  printf("%d␣%d␣%d\n",a,b,c);
}
```

```
$ ./assign
5 7 9
10 7 9
4 4 4
$
```

Attention piège

L'affectation est une expression... On peut donc l'utiliser comme expression dans un test...

Question

```
int a;
...
if (a=0) printf("bonjour\n");
else printf("au revoir\n");
```

Qu'affiche ce programme ?

Affectation composée

Les opérateurs v *op*= e se comportent exactement comme $v = v$ *op* e, mais produisent une expression dans laquelle la l-value n'est calculée qu'une seule fois. Ces opérateurs sont associatifs à droite.

Question

```
int a=1, b=54, c=2, d=12;
a += b /= c *= d += 1;
```

Que produit ce programme ?

Opérateur séquentiel ,

Definition (,)

L'opérateur , construit une expression qui est l'enchaînement ordonné de deux expressions, comme:

```
2*PI*radius, 4./3.*PI/pow(radius,3), (3+a)*x
```

La valeur de l'expression composée est celle de la dernière expression évaluée, ainsi:

```
a = 12, 34
```

affecte 34 à la variable `a`.

Instruction

Definition (Instruction «simple»)

Spécifie une action à réaliser.
Sont exécutées en séquence.
Syntaxe :

expression;

Exemple

```
13;
a;
a += b;
d = 1/2.*G*t*t;
```

Les instructions n'ont pas de valeur.

Bloc

Definition (bloc d'instructions)

Séquence d'instructions, sous la forme :

```
{
  items
}
```

où un *item* est soit une déclaration soit une instruction

Un bloc d'instructions est une instruction.

Exemple

```
{
  int i=0;
  int k;
  k = i+13;
  int j=k;
}
```


Sélection

Definition (Sélection)

Exécution d'une instruction sélectionnée en fonction de la valeur d'une expression de contrôle.

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

Pour le **if**, l'expression de contrôle doit produire une valeur de type scalaire (en gros les entiers et les flottants).
Pour le **switch**, l'expression de contrôle doit produire une valeur de type entier.

if

Si l'expression de contrôle est différente de `0`, c'est l'instruction qui suit qui est exécutée, sinon celle située derrière le `else` s'il existe. Le contrôle est ensuite rendu à l'instruction qui suit le bloc `if`.

Exemple

```
if (a==12) puts("la valeur est 12");
else       puts("La valeur n'est pas 12");
puts("suite...");
```

Note

L'expression `a==12` produit un résultat qui vaut 0 si a est différent de 12 et une autre valeur que 0 sinon.

switch

La valeur de l'expression de contrôle (un entier) :

- permet de sélectionner dans les instructions du bloc qui suit, la séquence à exécuter,
- la sélection consiste à rechercher un cas correspondant à la valeur, celui-ci doit être étiqueté par **`case`** `entier`**`:`**
- si aucun cas ne correspond et qu'il existe un cas **`default:`**, c'est celui qui est sélectionné,
- le saut est effectué et les instructions sont exécutées en séquence jusqu'à :
  - l'instruction **`break;`** qui permet d'obtenir la rupture de la séquence et le saut vers l'instruction qui suit le bloc **`switch`**,
  - ou la fin du bloc **`switch`**.

Exemple

```
switch (a) {
  case 3: puts("3");
  case 2: puts("2");
  case 1: puts("1");
          break;
  default:puts("bof");
}
puts("C'est␣terminé");
```

⚠ Attention

L'oubli du **break** est (très) fréquent et source d'importantes erreurs. Il est plus prudent d'en mettre pour chaque cas...

obfuscated C

L'écriture de code illisible est une discipline en soi, le terme adéquat pour le C est **obfuscated C** (ou C obscur) et fait l'objet d'un concours annuel : `ioccc.org`.
S'il n'est pas conseillé d'écrire un code «industriel» de ce type, cet art est au moins distrayant.
La syntaxe du C est assez permissive et autorise toutes sortes de constructions plus bizarres les unes que les autres.

Exemple

```
int a=0;
if (a=0); else puts("d'accord");
switch (a=5)   puts("d'accord");
switch(a=3)if(a=5)puts("3");else case 3:puts
    ("4");
```

L'utilisation d'une affectation comme contrôle d'une sélection est si fréquente et source de problèmes que le compilateur `gcc` émet un avertissement:

```
sw.c:11:8: warning: using the result of an assignment
     as a condition without parentheses [-Wparentheses
    ]
  if (a=0); else puts("d'accord");
       ~^~
sw.c:11:8: note: place parentheses around the
    assignment to silence this warning
  if (a=0); else puts("d'accord");
       ^
      (  )
sw.c:11:8: note: use '==' to turn this assignment
    into an equality comparison
  if (a=0); else puts("d'accord");
       ^
       ==
1 warning generated.
```


L'option de `gcc` correspondante est `-Wparentheses` et est activée par défaut.
Pour la désactiver (ne le faire que si strictement nécessaire), il faut utiliser `-Wno-parentheses`.

Note

Les options de `gcc` qui contrôlent la finesse des avertissements sont accessibles *via* `-W`.
L'option `-Wall` permet d'obtenir l'activation de tous les avertissements les plus critiques.
L'option `-Werror` permet de transformer tous les avertissements détectés en erreurs, de sorte qu'aucun code objet ne soit généré en cas d'avertissement (une erreur détectée par le compilateur empêche la production de code objet).

Question

Les programmeurs expérimentés utilisent l'idiome :

```
if (3==a) {
...
} else {
...
}
```

Pourquoi ?
Cela ne change rien à l'affaire si l'on «teste» deux variables :

```
if (b==a) {
...
```


Itération

Definition (Itération)

Exécution répétée d'une instruction et dont la répétition est conditionnée par une expression de contrôle (de type scalaire, *entier ou flottant*).
L'itération se termine lorsque l'évaluation de l'expression de contrôle est égale à 0.

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

Pour la boucle **for**, les expressions sont optionnelles, c'est-à-dire que l'expression vide est autorisée.

while

Exemple

```
int i=11;
while (i!=1) { // Syracuse(11)
  printf("i=%d\n",i)
  if (i%2==0) i /= 2;
  else        i  = 3*i+1;
}
```

Note

Les boucles infinies sont parfois écrites :

```
while (1) {
  ...
}
```


for

Exemple

```
for (int i=0; i<4; i++)   // ©Run-DMC
  puts("You talk too much...");
```

Les boucles **for** sont en réalité des **while** «déguisés» (on dit «du sucre syntaxique» — «syntactic sugar») :

Definition (for vs while)

for (e_1; e_2; e_3) i

e_1;
while (e_2) { i e_3; }

Exemple

```
for (int i=11; i!=1; ) { // Syracuse(11)
  printf("i=%d\n",i)
  if (i%2==0) i /= 2;
  else        i  = 3*i+1;
}
```

Note

Les boucles infinies sont parfois écrites :

```
for (;;) {
  ...
}
```

L'expression de contrôle vide est alors remplacée par une constante non nulle : `for(;1;)...`

Saut

Definition (Saut)

Rupture inconditionnelle de la séquence qui conduit à une instruction implicitement ou explicitement désignée.

Nous avons déjà étudié les sauts conditionnels avec la sélection (branchement/aiguillage en avant) ou l'itération (retour au commencement/branchement vers la suite).

goto

Instruction décriée et mal-aimée depuis :

Goto considered harmful
Edsger Wybe Dijkstra
Communications of the ACM 11
Mars 1968. pp. 147–148

Source : wikipedia

*The unbridled use of the **go to** statement has as an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. ... The **go to** statement as it stands is just too primitive, it is too much an invitation to make a mess of one's program.*

Il existe de bons usages du **`goto`**, seuls l'abus ou ses mauvais usages sont à proscrire...

En C, l'usage du **`goto`** repose sur la définition d'instructions étiquetées.

Definition (Instruction étiquetée)

Nommée par un identificateur :

```
identificateur: instruction
case constante: instruction
default:        instruction
```

Pour le premier cas, l'*identificateur* doit être unique dans la fonction qui le contient.
Les deux derniers cas ne peuvent apparaître que sous le contrôle d'une sélection **`switch`**.

Definition (`goto`)

```
goto identificateur;
```

Exemple (lisible, mais utile ?)

```
  for (int i=0; i<20; i++) {
    if (i==10) { goto END; }
    printf("%d\n",i);
  END: puts("coucou");
  }
```

Note

Les boucles infinies sont parfois écrites :

```
LOOP:
  ...
goto LOOP;
```


Idiome

```
if (!f()) goto end;
if (!g()) goto clean_f;
if (!h()) goto clean_g;
...
clean_g:
  ...
clean_f:
  ...
end:
  ...
```

Cet idiome est la version C du pattern RAII du C++ (vous verrez plus tard ?).

continue

Instruction (avec **break**) encensée car correspond à des usages raisonnés du **goto** :

*Structured programming with **go to** statements*
Donald Ervin Knuth
Computing surveys 6/4
Décembre 1974. pp. 261–301


Source : wikipedia

*There has been far too much emphasis on **go to** elimination instead of on the really important issues; people have a natural tendency to set up an easily understood quantitative goal like the abolition of jumps, instead of working directly for a qualitative goal like good program structure. . . .*

Il existe de bons usages du **goto**, **continue** et **break** en sont...

Definition (continue)

Ne peut apparaître que dans le corps d'une itération et permet de court-circuiter la séquence du corps pour passer à l'itération suivante :
continue;

```
while (...) {
  ...
  continue;
  ...
end: ;
}
```

```
do {
  ...
  continue;
  ...
end: ;
} while (...);
```

```
for (...) {
  ...
  continue;
  ...
end: ;
}
```

Pour chacun de ces usages, l'instruction **continue;** est équivalente à **goto** end;

Exemple

```
int sum = 0;
int prod = 1;
for (int i=0; i<20; i++) {
  n = getNumberFromKeyboard();
  if (n<0 && isPrime(n)) continue;
  sum += n;
  if (n%2==1) continue;
  prod *= n;
}
```


break

Definition (break)

Ne peut apparaître que dans le corps d'une sélection **switch** ou une itération et permet de rompre la sélection ou l'itération en se branchant immédiatement à l'instruction qui la suit :
break;

Exemple

```
int i = 0;
for (i=0; i<l; i++) {
  if (t[i]==0) break;
}
if (i!=l)  // found
else       // not found
```


Fonction (informatique)

Definition (Fonction)

Séquence d'instructions dédiée à l'accomplissement d'une tâche et constituée en une unité réutilisable.
On utilise aussi les termes : **sous-routine**, **routine**, **procédure**, **méthode**, ou encore **sous-programme**. Le terme générique d'**entité appelable** est parfois utilisée.

Une fonction est souvent le résultat d'une factorisation du code.

Conseil

Si vous êtes tenté de copier du code, il est urgent de songer à définir une fonction!

Fonction

La fonction C est caractérisée par :

- **un nom** qui doit être un identificateur
- **un corps** qui est la séquence d'instructions qui la constitue
- **des paramètres** qui sont des variables de la fonction dont les valeurs sont fixées à l'extérieur au moment de l'appel
- **un retour** qui est une variable cachée permettant de transmettre à l'appelant le résultat du calcul de la fonction sur ses paramètres

Definition (Fonction)

Une définition de fonction a la forme :

```
type identificateur([type identificateur, ...]) {
  ...
}
```

Une fonction peut ne pas avoir de paramètres, auquel cas la liste doit être égale à **`void`**.
Une fonction peut ne pas renvoyer de valeur, auquel cas le type de retour doit être **`void`**.

void

Definition (Le type `void`)

Le type `void` est un type incomplet (nous n'insisterons pas plus) et qui ne possède **aucune** valeur, par conséquent il ne peut exister de variables de ce type.

⚠ Attention

Le type `void *` fonctionne autrement (voir le chapitre sur les pointeurs).

Exemple de définition

```c
#include <stdio.h>

// définition de la fonction
int somme(int n) {
  int s = 0;
  for (int i=0; i<n; i++) {
    s += i;
  }
  return s;
}

int main(int argc, const char *argv[]) {
  // instruction faisant appel à la fonction définie
  printf("%d\n",somme(42));
}
```

La compilation est obtenue par :

```
$ gcc -Wall -o somme somme.c
$
```

L'exécution produit :

```
$ ./somme
861
$
```


Une tentative de modifier l'ordre des définitions :

Exemple de définition

```c
#include <stdio.h>

int main(int argc, const char *argv[]) {
  // instruction faisant appel à la fonction définie
  printf("%d\n",somme(42));
}

// définition de la fonction
int somme(int n) {
  int s = 0;
  for (int i=0; i<n; i++) {
    s += i;
  }
  return s;
}
```

La compilation est obtenue par :

```
$ gcc -o sommebad sommebad.c
sommebad.c:5:17: warning: implicit
   declaration of function 'somme' is
   invalid in C99 [-Wimplicit-function-
   declaration]
  printf("%d\n",somme(42));
                ^
1 warning generated.
$
```

Le compilateur se plaint car la fonction est utilisée **avant** d'être définie...

Mais un exécutable est tout de même généré qui produit :

```
$ ./somme
861
$
```

Prototype

Le compilateur avait averti d'un problème car pour qu'il puisse vérifier que l'appel était syntaxiquement correct (que les arguments passés sont bien en bon nombre et d'un type compatible avec les paramètres déclarés), la fonction doit avoir fait l'objet d'une **déclaration**.
La **définition** d'une fonction fait office de **déclaration**.

Definition (Prototype)

Une déclaration de fonction consiste en la définition de son **prototype**, c'est-à-dire ce qui est strictement nécessaire pour vérifier la correction des appels, donc le type de retour, l'identificateur, les types (et éventuellement les identificateurs) des paramètres.

```
type identificateur([type [identificateur],
   ...]);
```


Exemple

```
#include <stdio.h>
int somme(int); // déclaration / prototype
int main(int argc, const char *argv[]) {
  printf("%d\n",somme(42)); // appel
}
int somme(int n) { // définition
  int s = 0;
  for (int i=0; i<n; i++) { s += i;  }
  return s;
}
```

Note

Il doit y avoir une correspondance stricte entre un prototype et la définition correspondante!

Note

Dans le prototype `int somme(int);` le nom des paramètres n'apparaît pas. Ce n'est pas nécessaire puisque les informations présentes suffisent à vérifier la correction des appels.
Toutefois, les faire apparaître peut être utile, cela fait partie de la **documentation** du code; nommer les paramètres permet à la lecture du prototype de deviner leur usage, `int somme(int upperBound);`.
Il n'y a aucune nécessité à ce que le nom d'un paramètres dans un prototype corresponde au nom du même paramètre dans la définition! Dans le prototype c'est de la documentation, dans la définition c'est du code...

Note

Les directives `#include` servent à «inclure» dans un code source les déclarations de fonctions de bibliothèques. Pour chaque fonction «standard», le manuel indique quel fichier est à inclure.

Prototype (fonction sans paramètres)

Si l'on souhaite **définir une fonction sans paramètres**, il suffit que la liste de ses paramètres soit vide (il est possible d'indiquer simplement `void`). Une telle fonction, après sa définition, ne peut être appelée avec des arguments.
Si l'on souhaite **déclarer une fonction sans paramètres**, il faut se méfier :

- (ANSI-C) si la liste des paramètres est réduite à `void`, c'est le prototype standardisé d'une fonction sans paramètres.
- (K&R) si la liste des paramètres du prototype est vide, il s'agit d'un prototype K&R (autorisé dans les C modernes) ce qui signifie que la fonction est déclarée mais pas la liste de ses arguments (attention aux appels, qui eux doivent être conformes à la définition!).

Transmission de paramètres

Les paramètres d'une fonction font partie des ses variables locales. Mais leur initialisation est réalisé à l'appel :

- à l'appel, l'appelant (*caller*) fournit des valeurs,
- celles-ci servent à initialiser les paramètres (variables locales) correspondant,
- et le contrôle est transféré à la première instruction du corps de la fonction appelée (*callee*),
- durant l'exécution les paramètres se comportent comme n'importe quelles variables...

Definition (Mode de transmission en C)

Ce mode de transmission est appelé **transmission par valeur** ou **passage par valeur** ou encore **par copie**.

Attention

Quoique vous puissiez lire par ailleurs, c'est le seul mode de transmission disponible en C!
Il n'y en a pas d'autre!
Non!
NON!

Je sais... Le cas des pointeurs sera étudié plus tard...mais ils n'échappent pas à la transmission par valeur!

Note

La transmission de la valeur de retour est aussi réalisée par valeur.

`return`

Definition (`return`)

Le mot-clé **`return`** permet de définir une instruction de retour d'appel. Lorsque la fonction a pour type de retour `void` l'instruction ne doit pas comprendre d'expression. Lorsque la fonction a pour type de retour le type *T*, `return` doit être suivi par une expression compatible avec le type *T*.
Cette instruction termine immédiatement l'exécution du bloc de la fonction et renvoie à son point d'appel et l'expression d'appel prend la valeur (si elle existe) de l'expression du **`return`**.

Note

`return` n'est pas une fonction... On écrit **`return`** `bidule;` et non **`return`**`(bidule);`.

Old Times, Good Times
— Stephen Stills —

Fonction en K&R

Si jamais vous avez la chance (?!) de tomber sur un vieux code (type K&R), alors vous devez savoir que :

- le concept de prototype n'y existait pas vraiment...
- une fonction non déclarée était implicitement déclarée comme renvoyant un **`int`**
- la définition d'une fonction ne faisait apparaître que les identificateurs des paramètres, le typage était externe à la liste comme dans :

```
int add(a,b)
     int a;
     int b;
{
  return a+b;
}
```


du mode d'existence des variables

Techniquement, le standard ne définit pas la notion de variable. Il y existe le concept d'identificateur et celui d'objet (qui ne doit pas être entendu tel que dans le monde de la POO). Le terme objet réfère, en C, à celui de contenant (une zone mémoire pour faire court). Un identificateur permet de désigner un contenant.
Trois questions naturelles les concernent:

1. comment vivent les objets? On parle de **durée de vie** de la variable.
2. comment vivent les identificateurs? On parle de **portée** de l'identificateur ou de **visibilité** de la variable.
3. comment un identificateur est-il lié à un objet? On parle de **liaison**.

Durée de vie

Definition (Durée de vie)

Portion de l'exécution durant laquelle, l'existence de l'objet est garantie. D'autre part, durant toute sa vie, un objet possède une adresse constante (un emplacement en mémoire) et a la capacité de retenir la dernière valeur qui y a été stockée.

Il existe en C quatre types de durée de vie pour un objet: **statique** (*static*), **thread** (*thread*), **automatique** (*automatic*) et **dynamique** (*allocated*). On oubliera la durée de type thread (reportée à un autre cours?).

Durée statique

Definition (Durée de vie statique)

Possède une telle durée toute variable globale. Cette **durée** s'étend sur toute l'**exécution du programme**. Il est important de noter que de telles variables sont initialisées (toujours) au démarrage de l'exécution du programme.

Definition (variable statique ou globale)

Toute variable qualifiée par le mot-clé `static` ou déclarée en dehors de tout bloc est une **variable statique** (ou **globale**).

Exemple

```c
#include <stdio.h>

static int glob_in_source = 666;
int glob = 999;

void g() {
  printf("globale␣%d\n",glob_in_source);
  printf("globale␣%d\n",glob);
  glob_in_source++;
  glob++;
}

void f() {
  static int glob_in_f = 12;
  printf("globale␣(f)␣%d\n",glob_in_f);
  glob_in_f++;
  printf("globale␣%d\n",glob_in_source);
  printf("globale␣%d\n",glob);
  glob_in_source++;
  glob++;
}

int main(int argc, char *argv[]) {
  f(); g(); f(); g();
}
```


Exemple


```c
#include <stdio.h>

static int glob_in_source = 666;
int glob = 999;

void g() {
  printf("globale␣%d\n",glob_in_source);
  printf("globale␣%d\n",glob);
  glob_in_source++;
  glob++;
}

void f() {
  static int glob_in_f = 12;
  printf("globale␣(f)␣%d\n",glob_in_f);
  glob_in_f++;
  printf("globale␣%d\n",glob_in_source);
  printf("globale␣%d\n",glob);
  glob_in_source++;
  glob++;
}

int main(int argc, char *argv[]) {
  f(); g(); f(); g();
}
```


```
$ gcc -Wall -o  glob glob.c
$ ./glob
globale (f) 12
globale 666
globale 999
globale 667
globale 1000
globale (f) 13
globale 668
globale 1001
globale 669
globale 1002
$
```

Durée automatique

Definition (Durée de vie automatique)

Possède une telle durée toute variable locale (paramètre, variable locale). Cette **durée** s'étend sur toute l'**exécution du bloc** qui contient sa définition. Il est important de noter que sauf mention contraire, ces variables ne sont pas initialisées par défaut.
Une variable a une durée de vie automatique si elle n'est pas globale. Elle peut être qualifiée par le mot-clé `auto`.

Durée automatique

⚠ Attention

Il faut toujours prendre soin d'initialiser les variables automatiques, c'est la source de très nombreux bogues.

Le **temporaire** n'apparaît que lors de l'évaluation d'expressions.

⚠ Attention

Il faut prendre soin de ne jamais garder un quelconque accès à une variable automatique, *via* un identificateur dont la portée dépasserait celle de la variable automatique.
Autrement dit, toute tentative d'accès à une variable automatique en dehors de sa durée de vie conduit à un **comportement non défini** (*undefined behavior*).

Undefined behavior

Definition (Undefined behavior)

Comportement d'un programme qui serait syntaxiquement correct mais logiquement incorrect. L'échelle des comportements non définis va d'une exécution erratique pour cause de données inconsistantes à l'arrêt brutal de l'exécution.

Exemple

```
#include <stdio.h>

void g() {
  auto int local_in_g = 0;
  printf("locale (g) %d\n",local_in_g);
  local_in_g++;
}

void f() {
  int local_in_f = 0;
  printf("locale (f) %d\n",local_in_f);
  local_in_f++;
}

int main(int argc, char *argv[]) {
  f(); g(); f(); g();
}
```

```
$ gcc -Wall -o auto auto.c
$ ./auto
locale (f) 0
locale (g) 0
locale (f) 0
locale (g) 0
$
```


Portée d'un identificateur

Definition (Portée)

Portion du programme dans laquelle l'identificateur est visible.

Il existe quatre types de portée en C : **fonction**, **fichier**, **bloc** et **prototype**.

Note

Un même identificateur peut désigner plusieurs choses dans un même programme, à condition que cet identificateur serve à désigner des choses qui ne soient pas de même portée.
Deux identificateurs ont la même portée, si elles se terminent au même point du programme.

Portée fonction

Il n'existe qu'une seule construction pour cette portée : les étiquettes d'instructions.

Portée fichier

Tout identificateur dont la déclaration est faite en dehors de tout bloc a la portée du fichier.

Portée bloc

Tout identificateur dont la déclaration est faite à l'intérieur d'un bloc a la portée de ce bloc. Les paramètres déclarés d'une fonction ont la portée du bloc qui constitue le corps de la fonction.

Note

Un même identificateur peut désigner deux objets différents et l'une des portées inclure le domaine de l'autre. Dans ce cas, la variable dont la portée englobe le domaine inclus est masquée.

Portée prototype

Tout identificateur utilisé pour associer un identificateur à un paramètre a la portée du prototype qui le contient.

Liaison

Il existe trois types de liaisons:

1. externe
2. interne
3. sans liaison

Dans un programme constitué de différents codes, toutes les déclarations d'un même identificateur à liaison externe correspondent au même objet.
Dans un fichier de code source, toutes les déclarations d'un même identificateur à liaison interne correspondent au même objet.
Les déclarations d'identificateurs sans liaison correspondent à des objets uniques.

Identificateur sans liaison

Definition

Est **sans liaison** tout identificateur qui :

- se réfère à autre chose qu'un objet ou une fonction,
- ou correspond à un paramètre de fonction,
- ou a pour portée le bloc et ne correspond pas à un objet qualifié par **`static`**.

Identificateur à liaison interne

Definition

Est à **liaison interne** tout identificateur qui correspond à un objet ou une fonction de portée fichier et est qualifié par **`static`**.

Identificateur à liaison externe

Definition

Est à **liaison externe** tout identificateur:

1. correspondant à une fonction non qualifiée par **`static`**,
2. d'objet de portée fichier non qualifié par **`static`**,
3. d'objet qualifié par **`extern`** (sauf s'il existe déjà une déclaration à liaison interne du même identificateur).

Note

La liaison des variables à liaison interne ou sans liaison est résolue par le **compilateur**.
Les liaisons externes sont résolues par l'**éditeur de liens**.