Programmation Bas Niveau

-

Langage C

Aurélien Bossard

Table des matières

Introduction

Ce cours a pour objectif de vous familiariser à la programmation bas niveau, donc à des langages qui sont au plus proche du fonctionnement de la machine. Les langages qui se rapprochent le plus de la machine sont les langages dits « machine » – les langages natifs des processeurs, en binaire – et les assembleurs – langages machine lisibles par un humain.

Les langages de bas niveau s'opposent aux langages de haut niveau, qui permettent de s'abstraire de la couche matérielle, voire pour certains de la couche système, et de s'approcher au plus de la pensée humaine. Parmi les langages de haut niveau, on peut citer Java, PHP, CamL... Dans la famille des langages de haut niveau, certains sont compilés, d'autres semi-compilés ou interprétés. La différence entre langages de haut et de bas niveau ne se situe certainement pas dans le fait qu'ils soient ou non compilables.

Ici, nous nous intéressons à un langage majoritairement considéré comme de bas niveau : le langage C. Celui-ci se place au-dessus des assembleurs et des langages machine, puisqu'il permet de s'abstraire en partie du fonctionnement précis de la machine.

0.1 Historique

Le langage C a été inventé par Dennis Richie et Ken Thompson en 1972. Dans le même temps, Ken Thompson et Dennis Ritchie développaient UNIX. Le langage C et les systèmes UNIX sont donc étroitement liés.

En 1983, l'Institut national américain de normalisation (ANSI) a commencé à normaliser le langage C. Ce travail de normalisation a abouti en 1989 à la norme C ANSI (C89). Cette norme a par la suite été adoptée par l'Organisation internationale de normalisation en 1990 (ISO/CEI 9899 :1990 ou C90).

Le langage C va connaître deux évolutions majeures, l'une en 1999 (C99) et l'autre en 2011 (C11). Le compilateur gcc sous UNIX, entre autres, permet de compiler un programme en utilisant l'une ou l'autre de ces différentes normes.

0.2 Généralités

Le C est un langage compilé. Il existe d'autres catégories de langage utilisant des stratégies d'exécution différentes : la semi-compilation, la compilation à la volée et l'interprétation.

La compilation permet de maximiser les performances d'un programme en l'adaptant au système sur lequel il sera exécuté. Cela a cependant pour effet de spécialiser le programme : celui-ci ne peut être exécuté que sur le système pour lequel il a été compilé.

Un programme C est donc décrit par un fichier texte : la source. Ce fichier n'est pas compréhensible par le système sur lequel on veut l'exécuter. Il faut par conséquent passer par une phase de compilation des sources afin d'obtenir un fichier exécutable.

0.3 Normes d'écriture

En C, une commande se termine systématiquement par un « ; ».

Les blocs de code (corps des fonctions, code exécuté par les branchements conditionnels ou par les boucles...) sont délimités par { et }. Dans un bloc de code, par un souci de lisibilité, le code doit être indenté, c'est-à-dire aligné sur une même colonne en avant par rapport au signe de début de bloc. Certaines conventions existent pour l'indentation (tabulations, nombres d'espace...) mais l'indentation est libre du moment qu'elle est régulière et aide à la lisibilité.

Il est possible de placer des commentaires dans le code. Les lignes commentées seront ignorées lors de la compilation. Une ligne est commentée si elle commence par « // ». Pour faire des commentaires multi-lignes, il faut entourer les lignes à commenter de « /* » et « */ ». Exemple :

```
1 //Ceci est une ligne commentée
2
3 /*Ceci est un commentaires
4 multi-lignes.
5 Il s'arrête à la prochaine
6 ligne
7 */
```

Les noms de variable et de fonction doivent commencer par une miniscule, tandis que les noms de constante doivent être entièrement écrits en majuscules.

Chapitre 1

Premier programme et compilation

1.1 Un programme minimal

1.1.1 Inclusion d'en-têtes

Les en-têtes sont des fichiers qui contiennent des définitions de fonctions et de constantes. Il en existe deux types : les en-têtes de la bibliothèque standard et les autres. Nous y reviendrons plus tard dans le chapitre sur la modularité. Lorsque l'on veut utiliser une fonction déclarée dans un fichier d'en-tête, ce fichier d'en-tête doit obligatoirement être inclus au début du programme. Pour inclure une en-tête, on utilise le mot clé **#include**, suivi du nom du fichier d'en-tête entre chevrons s'il s'agit d'un fichier en-tête de la librairie standard, entre "" sinon.

Ainsi, pour inclure l'en-tête *stdlib.h*, dont nous aurons besoin pour notre programme minimal, nous écrivons au début du programme :

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

Ce fichier d'en-tête contient des fonctions très utiles, ainsi que la définition de plusieurs constantes, dont la constante EXIT_SUCCESS, dont nous nous servirons dans le programme.

1.1.2 Fonction main

La fonction main est une fonction obligatoire dans un programme compilable. Elle contient le code qui sera exécuté lors de l'exécution du programme, une fois celui-ci compilé.

Comme toute fonction, la fonction main contient un type de retour ainsi que des paramètres. Par défaut, le type de retour de la fonction main – c'est-à-dire le type de la valeur qui sera renvoyée au programme appelant – est *int*. Comme toute fonction également, son corps (c'est-à-dire le code qui la compose) est

délimité par des { } qui sont donc indispensables. L'utilisation du mot clé return met fin à la fonction main, donc au programme. L'utilisation de la fonction exit ou que ce soit dans le programme met également fin au programme.

Les paramètres peuvent varier : soit le programme ne prend aucun paramètre en entrée, auquel cas dans un souci de rigueur, le paramètre déclaré sera *void* ; soit le programme prend en entrée des paramètres bien définis, auquel cas les paramètres seront listés, soit le programme prend un ou plusieurs paramètres en entrée sans que tous soient obligatoires (c'est le cas de la majorité des commandes systèmes), et dans ce cas les paramètres seront *int argc, char* argv[]*[1]. Le premier paramètre correspond au nombre de paramètres passés au programme par le programme appelant, le deuxième aux paramètres eux-mêmes.

Les trois prototypes possibles pour la fonction main sont donc :

```
1  int main (void)
2  int main ( /*paramètres clairement définis*/)
3  int main (int argc, char* argv[])
```

La fonction main est déclarée avec une valeur de retour de type *int*. Par conséquent, il est impératif qu'elle renvoie une valeur au programme appelant. Cela se fait avec le mot clé **return**. L'en-tête *stdlib* définit plusieurs constantes symboliques[2] dont deux qui nous sont utiles ici :

EXIT_SUCCESS	à utiliser en cas de succès de l'application
EXIT_FAILURE	à utiliser en cas d'échec de l'application

Notre programme minimal est donc le suivant :

```
1  #include<stdio.h>
2
3  int main(void)
4  {
5    //Tout (rien en fait) s'est bien passé, on renvoie succès.
6    return EXIT_SUCCESS;
7  }
```

1.2 Compilation

Le langage C est un langage compilé ; un programme C ne peut donc s'exécuter sur un système donné que lorsque ses sources ont été compilées pour ce système. La compilation consiste à traduire des sources, compréhensibles par un humain et le compilateur, en langage machine. La compilation se déroule en quatre étapes :

1 : Traitement par le pré-processeur ;
2 : Compilation ;
3 : Assemblage ;
4 : Edition de lien.

1. Nous expliquerons dans le chapitre **??** la signification du dernier paramètre.
2. *cf* Chapitre 2

1.2.1 Préprocesseur

Le préprocesseur exécute des traitements nécessaires au compilateur pour comprendre le programme. Il n'affecte ni ne vérifie la structure des sources.

Le préprocesseur remplace notamment les digraphes et trigraphes par les caractères correspondant. Il regroupe en une même ligne les lignes séparées physiquement par des sauts de ligne mais qui font partie de la même ligne logique. Il tokenise le fichier source (découpe en jetons et espaces insécables le code) et retire les commentaires. Il étend les macros de précompilation (remplace chaque appel à une macro par sa définiction).

1.2.2 Compilation

Le code source fourni par le préprocesseur est traduit par le compilateur en code assembleur, une série d'instructions propres à chaque micro-processeur.

1.2.3 Assemblage

L'assemblage transforme le code assembleur en un code binaire directement interprétable par le micro-processeur. Le fichier produit est appelé fichier objet.

1.2.4 Edition de liens

Même notre programme minimal utilise un fichier d'en-tête. En effet, nous utilisons la constante EXIT_SUCCESS, définie dans L'édition de liens consiste à lier entre eux les différents fichiers objets créés lors de l'opération d'assemblage. Après l'édition de liens, un fichier exécutable est généré.

1.2.5 Compilation avec gcc

gcc est un ensemble de compilateurs créés par le projet GNU. gcc est un logiciel libre. Il est capable de compiler plusieurs langages de programmation, bien que nous ne l'utiliserons dans le cadre de ce cours uniquement pour compiler du C.

Un appel simple au compilateur gcc se fait à l'aide de la commande suivante :

gcc -o chemin_du_fichier_en_sortie chemin_du_ou_des_fichiers_c

Cette commande sert à réaliser un exécutable à partir d'un ou plusieurs fichiers source. En l'absence de chemin de fichier de sortie, le résultat du compilateur sera écrit dans un fichier nommé a.out.

Le fichier source à compiler doit avoir l'extension « .c ».

Pour parcourir les dossiers, on utilise la commande cd nom_du_dossier. ls pour visualiser les fichiers et dossiers du dossier courant.

Une fois le fichier compilé, on l'exécute comme ceci : ./chemin_du_fichier_en_sortie

Si l'exécution échoue, peut-être les droits en exécution n'ont-ils pas été attribués au fichier compilé. Dans ce cas, vous pouvez les lui donner avec la commande :

chmod 755 chemin_du_fichier_en_sortie

Par défaut, gcc réalise une compilation complète avec des options standard. Cependant, il est possible d'isoler certaines étapes de la compilation, et d'utiliser des options de compilation avancées. Voici une liste non exhaustive d'options de gcc :

-o précise le fichier de sortie ;
-E active uniquement le préprocesseur ;
-S active uniquement le préprocesseur et le compilateur ;
-c n'exécute pas l'édition de liens ;
-std= détermine le standard à utiliser (c89, c99, ...). Voir la liste des valeurs possibles sur le man de gcc ;
-O, -O1, -O2, -O3 utilisent différentes stratégies d'optimisation de la compilation ;
-W affiche des messages d'avertissement supplémentaires ;
-Wall affiche tous les messages d'avertissement.

Chapitre 2

Variables / Constantes

2.1 Les Variables

2.1.1 Définition

Les variables associent un nom (ou symbole) à une valeur. La valeur d'une variable peut changer au cours de l'exécution d'un programme. Les variables sont des identifiants. Par conséquent leur nom doit être compris dans un espace de nommage bien défini.

En C, une variable a plusieurs caractéristiques :

- Son nom : un moyen d'accéder à la valeur de la variable ;
- Son type : la taille en mémoire de la variable ainsi que la façon d'interpréter sa valeur en mémoire ;
- Son adresse : l'endroit en mémoire où est stockée la variable ;
- Sa valeur : la suite de bits contenue à l'adresse de la variable ;
- Sa portée : la portion de code dans laquelle la variable existe.

2.1.2 Nommage

Les noms de variables doivent être compris dans l'espace de nommage du langage C. Les noms de variables sont sensibles à la casse. Les compilateurs C ne prennent en compte que les 32 premiers caractères du nom d'une variable afin de l'identifier.

Espace de nommage

En C, comme dans la plupart des langages, les noms de variable doivent commencer par une lettre ou un _, et être suivis de lettres, chiffres, ou _ . La traduction de l'espace de nommage en expression régulière est la suivante :

```
[a-zA-Z\_\x7f-\xff][a-zA-Z0-9\_\x7f-\xff]*
```

. Les noms de variables ne doivent pas rentrer en conflit avec les mots réservés du langage, dont voici la liste :

auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while.

FIGURE 2.1 – Mots clés réservés du langage C

Adresse en mémoire

Il est possible de connaître l'adresse en mémoire d'une variable. Pour cela, on utilise le symbole &, placé devant la variable. Par exemple, si *var1* est une variable, alors *&var1* est son adresse en mémoire.

Conventions

Par convention, les noms de variable ne commencent pas par une majuscule, et ne contiennent pas majoritairement des majuscules, afin d'éviter de les confondre avec les constantes symboliques (qui elles, sont par convention entièrement en majuscules). Par convention également, lorsqu'un nom de variable est l'association de plusieurs mots, il est préférable d'identifier le début de chaque mot avec une majuscule. Exemple : *idProgramme* plutôt que *idprogramme*.

2.1.3 Type

Le langage C fournit des types de variable dits « fondamentaux » :
- char : pour encoder les caractères en code ASCII ;
- int : pour encoder les entiers ;
- float : pour encoder les nombres flottants (à virgule) à précision simple ;
- double : pour encoder les nombres flottants à précision double.

Ces types ont une longueur en mémoire variable selon le système. À ces types fondamentaux peuvent être adjoints des modificateurs : *signed* (par défaut), *unsigned*, *short*, *long*, *long long*. L'emploi d'un modificateur sans adjonction d'un type fondamental équivaut à utiliser le modificateur suivi d'*int*. Le standard C99 ne définit pas de représentation obligatoire pour les nombres négatifs. Cela est motivé par le fait que la meilleure représentation diverge selon l'architecture ainsi que le système d'exploitation utilisés. Le standard C99 définit seulement la plus petite valeur obligatoirement prise en compte pour chaque type. Celle-ci est calculée d'après la représentation en complément à un. Voici un récapitulatif des types, de leur longueur en mémoire et de leur plage de valeur :

2.1.4 Déclaration

En C, la déclaration d'une variable est nécessaire à son utilisation. Une variable a pour portée – e.g. l'espace de code dans laquelle elle existe – le bloc de code dans laquelle elle a été déclarée. Un bloc de code est un code entouré de { et }.

Type	Longueur en mémoire (en bits)					Plage de valeur (minimum)
	C std	LP32	ILP32	LLP64	LP64	
short int	min 16	16	16	16	16	-32767,+32767
unsigned short int						0,35655
int	min 16	16	32	32	32	
7-7 unsigned int						
long int	min 32	32	32	32	32	-2147483647,+2147483647
float	min 16	16	32	32	32	
7-7 unsigned float						
double						
long double						

TABLE 2.1 – Types fondamentaux en C : longueur en mémoire, plage de valeur

Pour déclarer une variable, il suffit de préciser son type ainsi que son nom. Il est également possible d'affecter une valeur à une variable lors de sa déclaration. Ex :

```
1 int compteur = 0;
2 int uneAutreVariable;
```

2.1.5 Affectation

L'opérateur d'affectation est le signe =. Les opérateurs = et == ne doivent en aucun cas être confondus. L'un correspond à l'affectation, l'autre au test d'égalité entre les opérandes gauche et droite.

L'opérateur d'affectation a pour opérande gauche la variable à laquelle on veut affecter une valeur, à droite sa valeur. Il est possible d'utiliser comme valeur de droite une opération comportant la variable elle-même, le calcul de l'opération à droite de l'opérateur = étant prioritaire sur l'affectation elle-même. Ex :

```
1 int a;
2 int b = 1;
3 b = b + 1;
4 a = 2;
```

2.1.6 Opérateurs

Le langage C définit des opérateurs de base. En voici une liste non exhaustive :

Opérateurs de calcul

+ : addition réelle ou entière ;
- : soustraction réelle ou entière ;

/ : division réelle ou entière ;
* : multiplication réelle ou entière ;
% : modulo (reste de la division entière).

Opérateurs d'affectation

= : affecte l'opérande de droite à l'opérande de gauche ;
+= : affecte l'opérande de gauche plus l'opérande de droite à l'opérande de gauche ;
-= : affecte l'opérande de gauche moins l'opérande de droite à l'opérande de gauche ;
*= : affecte l'opérande de gauche fois l'opérande de droite à l'opérande de gauche ;
/= : affecte l'opérande de gauche divisé par l'opérande de droite à l'opérande de gauche ;
%= : affecte l'opérande de gauche modulo l'opérande de droite à l'opérande de gauche.

Opérateurs bit à bit

Les opérateurs bit à bit permettent de réaliser des opérations sur les données des variables en binaire.

& : ET logique bit à bit ;
| : OU logique bit à bit ;
^ : OU logique inclusif bit à bit ;
~ : NON logique bit à bit.

Opérateurs de décalage

Les opérateurs de décalage travaillent sur les données des variables en binaire. Ils permettent de décaler les bits d'une variable à droite ou à gauche. Par conséquent, ils reviennent à multiplier ou à diviser par des puissances de 2. Les bits dépassant de l'espace de stockage d'une variable sont supprimés.

« : décalage à gauche de l'opérande de gauche d'un nombre de cases défini par l'opérande de droite ;
» : décalage à droite de l'opérande de gauche d'un nombre de cases défini par l'opérande de droite.

Opérateurs de comparaison

Les opérateurs de comparaison renvoient 0 si la comparaison est fausse, 1 sinon.

== : comparaison d'égalité. Attention à ne pas le confondre avec l'opérateur d'affectation, cause de beaucoup d'erreurs !
!= : inégalité ;
> : supériorité stricte ;
< : inferiorité stricte ;

>= : supériorité.
<= : infériorité.

Opérateurs logiques

Il n'y a pas de type booléen en C. Toute valeur égale à 0 vaut faux, et toute valeur différente de 0 vaut vrai. Les opérateurs logiques agissent sur les int, float et double.

&& : ET logique. A ne pas confondre avec le ET bit à bit ;
|| : OU logique. A ne pas confondre avec le OU bit à bit ;
! : NON logique.

2.1.7 Saisie par l'entrée standard

C fournit une fonction pour saisir des valeurs sur l'entrée standard et les stocker dans une variable. C'est la fonction scanf, définie dans le fichier en-tête <stdio.h>. Celui-ci doit donc être inclus grâce à la commande :

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

afin d'être utilisée dans un programme.

La fonction scanf fonctionne comme la fonction printf : elle prend un nombre dynamique de paramètres, dépendant du nombre de mots clés contenus dans le premier paramètre, une chaîne de caractère. Cette chaîne de caractère sert à caractériser le type, le nombre et l'ordre des variables à saisir sur l'entrée standard. Il est important de préciser à l'utilisateur qu'il est face à une saisie grâce à un affichage avec la fonction printf.

Les paramètres correspondant aux variables à saisir sont non pas les variables elles-même, mais leur adresse en mémoire. Pour obtenir l'adresse d'une variable, il faut placer devant son identifiant le symbole &. Ainsi, si a est une variable, &a est l'adresse en mémoire de la variable a. Voici des exemples de scanf :

```
1  int entier;
2  float flottant;
3  char car;
4
5  printf("Entrez une valeur entière : ");
6  scanf ("%d", &entier);
7  printf("\nEntrez une valeur réelle : ");
8  scanf ("%f", &flottant);
9  printf("\nEntrez un caractère : ");
10 scanf ("%c", &car);
```

La fonction scanf renvoie le nombre de variables qu'elle a affectée. Ceci est utile pour vérifier que les saisies d'un utilisateur sont correctes.

2.1.8 Impression sur l'entrée standard

Il est possible d'afficher la valeur d'une variable. Pour ce faire, on utilise la fonction printf qui possède un nombre de paramètres dynamique, dépendant du

Mot-clé	Type de variable
%d	int
%f	float/double
%L	long double
%c	char
%s	chaîne de caractères
%p	adresse mémoire

TABLE 2.2 – Liste non exhaustive de mots clés d'affichage de variable dans la fonction printf

nombre de variables à afficher. La fonction printf prend en premier paramètre obligatoire une chaîne de caractère, soit une suite de caractères entourée de " ". L'utilisation de mots clés spécifiques permet de demander l'affichage d'une variable. Pour chacun des mots clés présents dans la chaîne de caractères en paramètre de printf, il faut une variable correspondante en paramètre de printf, dans le même ordre que celui de la chaîne. Comme pour toutes les fonctions en C, les paramètres sont séparés par des virgules. Tous les mots clés d'affichage de variables commencent par %. En voici une liste non exhaustive :

Exemple d'affichages :

```
1 int entier = 1;
2 char caractere = 'c';
3 float flottant = 1.23;
4 char[] chaine = "chaîne de caractères";
5 printf("%d \n", entier);
6 printf("%c \n", caractere);
7 printf("%s \n", chaine);
8 printf("On peut faire des choses étranges : %d \n", caractere);
9 printf("On peut aussi afficher plusieurs variables : %d , %f \n", entier,
      flottant);
```

La fonction printf est définie dans le fichier en-tête <stdio.h>. Celui-ci doit donc être inclus grâce à la commande :

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

afin d'être utilisée dans un programme.

Au-delà des types de base que peut afficher la fonction printf, il est possible d'ajouter des modificateurs d'affichage. On peut par exemple afficher les nombres sur un nombre de digits défini, ou encore limiter l'affichage des float ou des double à un certain nombre de digits après la virgule, *etc.* Le tableau 2.3 présente certaines de ces options.

Mot-clé	Signification	Exemple
-	Justifié à gauche, complété à droite par des espaces.	"%-d"
+	Affiche le signe d'un nombre, qu'il soit positif ou négatif.	%+f
0n	Sur n caractères, éventuellement complétés par des 0.	%03d
n	Sur n caractères, éventuellement complétés par des espacse	%3d
%p	adresse mémoire	

TABLE 2.3 – Liste non exhaustive de mots clés d'affichage de variable dans la fonction printf

2.2 Constantes

2.2.1 Caractérisation

Les constantes sont des identificateurs qui ne pourront pas être modifiés au cours du programme. Elles sont définies une fois pour toutes au début du programme, que ce soit dans le fichier contenant le main ou dans les fichiers en-tête inclus. En C, les constantes peuvent être des entiers, des flottants, des caractères ou des chaînes de caractères.

2.2.2 Espace de nommage

L'espace de nommage des constantes est le même que celui des variables. Les noms de constantes doivent commencer par une lettre ou un _, et être suivis de lettres, chiffres, ou _ . La traduction de l'espace de nommage en expression régulière est la suivante :

```
[a-zA-Z\_\x7f-\xff][a-zA-Z0-9\_\x7f-\xff]*
```

.

2.2.3 Conventions

Par convention, les noms de constantes sont entièrement en majuscule. Pour séparer les mots lorsqu'un nom de constante est composé de plusieurs mots, on utilisera le signe _.

2.2.4 Définition

Pour définir une constante, on utilise le mot clé #define suivi du nom de la constante et de sa valeur. La ligne de définition d'une variable ne se termine pas par un point virgule.

```
1 #define CONST_PRIX 1.23
2 #define NOM_IUT "Montreuil"
3 +
```

Chapitre 3

Structures de contrôle

Le langage C définit des structures de contrôle. Ces structures, associées à une condition, permettent d'exécuter des parties différentes du code selon que l'opération est validée ou non. Nous définissons ici les conditions logiques, puis présentons les différentes structures de contrôle.

3.1 Conditions

Les conditions en C sont exprimées sous la forme d'opérations logiques. Les opérations logiques ne renvoient pas de bouléens [1] en C, contrairement à d'autres langages. Les résultats d'une opération logique sont exprimés sous la forme d'entiers : une valeur inférieure ou égale à 0 équivaut à faux, et une valeur supérieure ou égale à 1 équivaut à vrai. Les parenthèses servent à effectuer certaines opérations en priorité, afin d'« écraser » les priorités des opérateurs définies par le langage.

3.1.1 Opérateurs de comparaison

Test d'égalité (==)

L'égalité entre deux valeurs est testée grâce à l'opérateur double égal (==). Celui-ci ne doit en aucun cas être confondu avec l'opérateur d'affectation. En effet, il n'y a pas de type bouléen en C. Par conséquent, la valeur de retour de l'opérateur d'affectation (un entier) peut être intégrée à une opération logique sans que cela ne provoque d'avertissement par le compilateur. Attention donc à ne jamais confondre ces deux opérateurs

Autres opérateurs

- $>$: Teste si l'opérande de gauche est strictement supérieure à l'opérande de droite ;

1. Bouléen : vrai ou faux

- >= : Teste si l'opérande de gauche est supérieure ou égale à l'opérande de droite ;
- < : Teste si l'opérande de gauche est strictement inférieure à l'opérande de droite ;
- <= : Teste si l'opérande de gauche est inférieure ou égale à l'opérande de droite ;

3.1.2 Opérateurs logiques

Opérateur non (!)

L'opérateur ! renverse l'opérande de droite.

a	1	0
!a	0	1

TABLE 3.1 – Table de vérité de l'opérateur non

Opérateur et (&&)

L'opérateur && réalise l'opération ET logique.

&&	0	1
0	0	0
1	0	1

TABLE 3.2 – Table de vérité du ET logique

Opérateur ou (||)

L'opérateur || réalise l'opération OU logique.

||	0	1
0	0	1
1	1	1

TABLE 3.3 – Table de vérité du ET logique

3.1.3 Priorité des opérateurs

L'ordre de priorité décroissante (non exhaustif) des opérateurs est : opérateurs de comparaison, !, &&, ||.

3.2 Structures conditionnelles

3.2.1 If/then/else

La structure conditionnelle if/then/else permet de tester une condition, d'exécuter la partie du code située dans le then si celle-ci est vraie, la partie du code située dans le else si la condition est fausse.

La structure conditionnelle if/then/else se présente ainsi :

```
if (condition)
{
  /*
  Code à réaliser si la condition est vraie
  */
}
else
{
  /*
  Code à réaliser si la condition est fausse
  */
}
```

Le programme suivant :

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

int main (void)
{
  int a;
  scanf("Entrez une valeur : %d", &a);
  if (a <= 0)
  {
    printf("La valeur entrée est inférieure ou égale à 0 !\n");
  }
  else
  {
    printf("La valeur entrée est supérieure à 0.\n");
  }

  return EXIT_SUCCESS;
}
```

affichera « La valeur entrée est inférieure ou égale à 0 ! » si l'utilisateur entre une valeur négative ou nulle, « La valeur entrée est supérieure à 0. » si l'utilisateur entre une valeure strictement positive.

3.2.2 switch/case

La structure conditionnelle switch/case permet d'exécuter des codes différents selon des valeurs bien identifiées que peuvent prendre une variable. On ne

peut pas utiliser de conditions dans un switch/case.

Le switch case porte sur une seule variable, passée en paramètre du switch. Par convention, switch/case comprend un dernier cas, *default*, qui comprendra le code à exécuter lorsque la variable testée ne correspond à aucun cas cité au préalable. Le switch/case utilise une commande spécifique, *break*, pour sortir des différents cas identifiés. Cette commande est indispensable à la fin de chaque cas.

Voici la syntaxe du switch/case, suivie d'un exemple :

```
switch(variable)
{
 case valeur1 :
  /*
  Code pour valeur1
  */
  break;
 case valeur2 :
  /*
  Code pour valeur2
  */
  break;
 ...
 default :
  /*
  Code pour les cas non gérés
  */
}
```

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

int main(void)
{
  int a;
  scanf("Voulez-vous afficher une étoile ? (o/n) : %c", &a);

  switch(a)
  {
   case 'o':
    printf("*\n");
    break;
   case 'n':
    printf("J'affiche quand même une étoile... *\n");
    break;
   default:
    printf("Je n'ai pas compris votre réponse.\n");
  }
  return EXIT_SUCCESS;
}
```

3.3 Boucles

Les boucles consistent en des structures qui itèrent un code un nombre de fois qui peut être défini ou indéfini. Plusieurs types de boucles existent, qui ne sont par convention pas utilisés dans les mêmes contextes.

3.3.1 Boucle for

La boucle for est utilisée dans le cas où l'on connait le nombre d'itérations de la boucle, donc lorsqu'on itère sur un nombre fini d'éléments, sans que rien ne puisse venir contrarier les conditions de la boucle (erreur de lecture dans un fichier, intervention d'un utilisateur par exemple). La boucle for est caractérisée par une variable qui est initialisée à une valeur de départ, une condition d'arrêt sur cette même variable, ainsi qu'un pas sur cette variable (incrémentation, décrémentation, ...).

Voici la syntaxe de la boucle for :

```
for (initialisation_de_la_variable ; condition ; pas )
{
  /*
  Code à exécuter en boucle
  */
}
```

Un exemple concret :

```
//Affichage de la table de 3
int compteur;
for (compteur = 0; compteur < 10; compteur ++)
{
  printf("%d x 3 = %d\n", compteur, compteur*3);
}
```

3.3.2 Boucle do/while

La boucle do/while (faire tant que) est utilisée lorsque l'on veut que le code englobé dans la boucle soit exécuté avant de vérifier la condition du tant que. La syntaxe est la suivante :

```
do {
  /*
  Code sur lequel on veut itérer
  */
}while (condition);
```

Notez le point virgule après la fermeture de la condition, obligatoire pour un do/while.

Voici un exemple :

```c
int a = 0;
do {
  printf ("%d\n", a);
  a++;
}while (a < 3);
```

Cet exemple affiche :

```
0
1
2
3
```

3.3.3 Boucle while

La boucle while (tant que faire) est utilisée lorsque l'on veut que le code englobé dans la boucle soit exécuté une fois la vérification de la condition réalisée. La syntaxe est la suivante :

```c
while (condition) {
  /*
  Code sur lequel on veut itérer
  */
}
```

Notez que contrairement au do/while, il n'y a pas de point virgule pour marquer la fin du while.

3.3.4 Structures sans accolades

Il est possible de se passer des accolades pour délimiter le bloc d'une structure de contrôle. Dans ce cas, seule la commande qui suit sera contrôlée par la structure de contrôle. Ex :

```c
if (a == 0)
{
  /*
  Faire quelque chose
  */
}
else
  a = -1;
//Ce qui suit ne fait pas partie du else
...
```

3.3.5 Commande break

La commande break permet de sortir d'une structure de contrôle. Dans le cas où le break est situé dans un if/then/else, le break sert à sortir de la boucle qui englobe le if/then/else.

L'exemple suivant utilise une boucle infinie (while ()) dont on peut sortir à tout moment selon une saisie de l'utilisateur au clavier :

```
1  while ()
2  {
3    scanf("Est-ce qu'on continue à poser la même question indéfiniment ? (o
         /n) %d", &variable);
4    if (variable == 'o')
5      break;
6  }
```

3.4 Imbrication de structures

Il est possible d'imbriquer plusieurs structures de contrôle. Pour cela, il suffit d'inscrire une structure de contrôle à l'intérieur d'un bloc (soit entre les deux accolades) d'une autre structure de contrôle. Voici un exemple de if/then/else imbriqués. decision est une fonction. Nous verrons les fonctions plus tard dans le cours, bien que nous ayons déjà vu, avec le main, un exemple de définition de fonction.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define VERT 0
5  #define VEGETAL 2
6  #define ANIMAL 1
7  #define HOMME 0
8
9  void decision (int branche, int couleur)
10 {
11   if (branche == 0)
12   {
13     if (couleur == 0)
14     {
15       printf("L'individu est un homme et doit être sérieusement malade !\
             n");
16     }
17     else
18     {
19       printf("L'individu est un homme.\n");
20     }
21   }
22   else if (branche == 1)
23   {
24     if (couleur == 0)
25     {
26       printf("L'individu est un animal de couleur verte. Probablement une
               grenouille, un iguane, ou un serpent tropical.\n");
27     }
```

```
28      else
29      {
30        printf("L'individu est un animal.\n");
31      }
32    }
33    else if (branche == 2)
34    {
35      if (couleur == 0)
36      {
37        printf("L'individu est un végétal de couleur verte. Rien de plus
              normal\n");
38      }
39      else
40      {
41        printf("L'individu est un végétal.\n");
42      }
43    }
44    return void;
45  }
```

Ce type de if/then/else peut être traduit sous la forme d'un arbre de décision. Pour vous entraîner, écrivez l'arbre de décision correspondant à la fonction decision ci-dessus.

Il est évidemment possible d'imbriquer n'importe quel type de structure de contrôle dans n'importe quel autre type de structure de contrôle. Exemple :

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main(void)
5   {
6     int compteur;
7     int reponse;
8     while ()
9     {
10      for (compteur = 0; compteur < 10; compteur++)
11      {
12        printf("*");
13      }
14      printf("\n");
15      do
16      {
17        scanf ("J'aime afficher des lignes de dix étoiles. Voulez-vous que
              je continue à afficher des lignes de dix étoiles ? (o/n) %c", &
              reponse);
18      }while(reponse != 'o' && reponse != 'n');
19
20      if (reponse == 'n')
21        break;
22    }
```

```
23   return EXIT_SUCCESS;
24 }
```