

— Cours de C —

RIPOPÉE INITIATIVE

RYAN BOUCHOU

Sommaire

1	Fondements	1
1.1	Historique	1
1.2	La compilation	1
1.2.1	Pré-traitement	1
1.2.2	Analyse lexicale	1
1.2.2.1	Unités lexicales	1
1.2.2.2	Processus	3
1.2.3	Analyse syntaxique	3
1.2.4	Suite et fin...	4
1.2.5	Le compilateur GCC	4
1.2.5.1	Options	4
1.2.5.2	En pratique	4
1.3	Les expressions	4
1.3.1	Identificateur	4
1.4	Types de bases	5
1.4.1	Type entier	5
1.4.2	Type float	5
1.4.3	Type bool	5
1.4.4	Type caractère	6
1.4.4.1	Manipulation des caractères A faire	6
1.4.5	Type chaîne de caractère	6
1.5	Les opérateurs	7
1.5.1	Opérateur d'affectation	7
1.5.2	Opérateurs arithmétiques	7
1.5.3	Opérateurs de comparaison	7
1.5.4	Opérateurs booléens	7
1.5.5	Compléments	8
1.5.5.1	Opérateurs composés	8
1.5.5.2	Opérateurs d'incrémentatation	8
1.5.5.3	Opérateur de coercition	8
1.6	Instructions et structures de contrôle	8
1.6.1	Les variables	8
1.6.1.1	Déclaration	8
1.6.1.2	Portée	9
1.6.1.3	Évolution de la pile	9
1.6.2	Les fonctions	10
1.6.2.1	Définition	10
1.6.2.2	Déclaration	10
1.6.2.3	Appel	10
1.6.2.4	Compléments	11
1.6.3	Structures conditionnelles	11
1.6.3.1	L'alternative	11
1.6.3.2	Opérateur ternaire	11
1.6.3.3	Le branchement multiple	12
1.6.4	Les boucles	12
1.6.4.1	Tant que...	12
1.6.4.2	Pour	12
1.6.4.3	Faire ... Tant que	13
1.6.4.4	Passes-droit	13

2	Types composés	14
2.1	Les structures	14
2.1.1	Définition	14
2.1.2	Instanciation	14
2.2	Les énumérations	14
2.3	Les unions	14
2.4	Les tableaux	14
2.4.1	Déclaration	14
2.4.1.1	Déclaration seule	15
2.4.1.2	Déclaration et initialisation	15
2.4.2	Accès aux valeurs	15
2.4.3	Tableaux multidimensionnels	15
3	La mémoire	16
3.1	Les pointeurs	16
3.2	Le cas des tableaux	16
3.2.1	Tableaux multidimensionnels	16
3.3	Le cas des structures	16
4	Annexe	17
4.1	Boîte à outils	17
4.1.1	Test ici	17
4.1.1.1	Encore un autre	17
4.2	Représentation des réels	18
4.3	Calcul propositionnel	19

1 Fondements

1.1 Historique

Le C est un langage de programmation apparu dans les années 70, dérivant directement du langage B, et inventé par Dennis Ritchie et Kenneth Thompson. C'est un langage impératif, procédural et bas niveau. En conséquence de quoi, il permet d'ériger des séquences d'instructions qui offrent, entre autre par le biais des pointeurs et des accès aux primitives du système, une manipulation ample mais périlleuse du hardware.

Notons qu'une grande partie des processeurs sont de natures impératives ; subséquemment, il est naturel que les langages de programmations _ qui visent en outre à simplifier l'interaction homme/machine et le contrôle de celle-ci _ se rapprochent d'un tel paradigme.

1.2 La compilation

Le C est un langage compilé. Le code est préalablement édité dans un *fichier source* dont l'extension est *.c* avant d'être traduit en langage machine (assembleur) par le compilateur. On décrit ci-dessous les différentes étapes de la compilation.

1.2.1 Pré-traitement

Lors de cette phase de pré-traitement, le pré-processeur procède en une mise aux normes textuelles du fichier source.

- Remplacement des *tri-graphes*
- Raccordement des lignes qui forment une unité logique
- Retrait des commentaires
- Expansion des macros et exécution des directives spécifiques (inclusion d'autres fichiers...)

1.2.2 Analyse lexicale

Aussi appelée *tokenization*, elle procède en une conversion de chaînes de caractère en symboles appelés *token*.

1.2.2.1 Unités lexicales

La convention syntaxique ANSI exhibe les *jurons*/ *unités lexicales*/ *lexèmes*/ *tokens* suivant:

Token	Exemple
mot-clé	if extern char
identifiant	mafonction x y
constante	0 15.3 1e-4 'c' "coucou"
chaîne	"coucou" true
opérateur	*+-
ponctuation	();

Table 1: Unités lexicales du C

Remarque

┆ Toutes ces composantes lexicales sont détaillées par la suite.

1.2.2.2 Processus

Afin de procéder à l'analyse lexicale, et d'exhiber les tokens appartenant au langage de programmation, il est judicieux de se munir d'automates. Ce faisant, un automate fini permet d'opérer un **balayage** du code source afin de déterminer chacun des lexèmes qui le compose en leur adjoignant leur type.

Exemple

De façon concrète, nous pouvons concevoir un automate qui reconnaît les entiers générés par une expression rationnelle $([0' - 9']^*)$

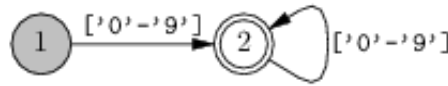


Figure 1: Automate reconnaissant les entiers [2]

Il faut ainsi imaginer qu'il en est de même pour le reste du langage.

Ensuite, l'**évaluation** convertit chaque lexème en une valeur. C'est aussi au cours de cette opération que les *espaces blancs* (à savoir les tabulations, les caractères espaces, les sauts de ligne...) sont supprimés.

Exemple

Considérons l'instruction `sum = 2 + 3;`. L'analyse lexicale produit une liste de lexèmes munis de leurs types, de telle sorte à être conforme à la **grammaires du langage** ainsi qu'à ses **regles de productions**.

```
1 | [(identifiant, sum), (opérateur d'affectation, =), (entier littéral, 2),  
  | ↪ (operator, +), (entier littéral, 3), (fin de déclaration, ;)]
```

1.2.3 Analyse syntaxique

La précédente analyse lexicale nous a permis de mettre en exergue un flux de lexèmes. Lors de l'analyse syntaxique ou grammaticale, on établit un arbre connexe acyclique qui décrit l'intrication syntaxique de notre code source en fonction de la grammaire (dite algébrique ou non contextuelle) du langage.

Exemple

Considérons l'expression $\phi = (x + 1) * (3 * y + 2)$. L'arbre de syntaxe abstraite représentant ϕ est le suivant:

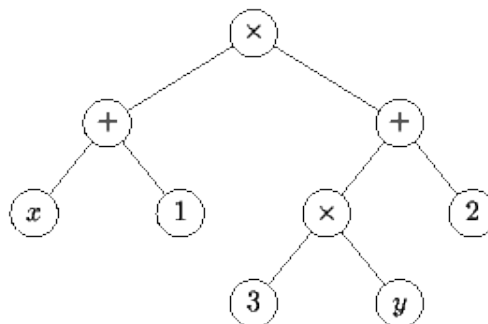


Figure 2: Arbre de syntaxe [1]

1.2.4 Suite et fin...

Suite à l'analyse sémantique, on obtient un code intermédiaire qu'il convient de transformer en code assembleur (langage machine). S'en suit un processus d'optimisation, d'allocation mémoire et d'édition des liens afin de produire, enfin, un fichier exécutable.

Remarque

Le détail du processus de compilation est plus riche et plus complexe que ce que cours mentionne. On se contente de ne donner que quelques éléments notionnels sur les grandes étapes du processus. Un cours complet de compilation repose sur la théorie des langages, la Sémantique Opérationnelle Structurale, une connaissance du hardware... Se référer à la bibliographie pour plus de détails.

1.2.5 Le compilateur GCC

Afin de compiler notre fichier source sous UNIX, on se munira du compilateur GCC développé par le projet GNU. L'appel au compilateur suit la syntaxe suivante:

```
1 | gcc [options] fichier.c [-llibrairies]
```

1.2.5.1 Options

- `-c` : Stoppe la compilation au stade du fichier objet.
- `-o nom-de-fichier` : Renommage du fichier produit. Par défaut, le exécutable fichier s'appelle `a.out`.
- `-Wall` : Affiche l'ensemble des messages d'erreur.
- `-O`, `-O1`, `-O2`, `-O3` : options d'optimisations, de la plus faible à la plus importante.

1.2.5.2 En pratique

Admettons que nous ayons codé notre programme dans le fichier `mon_Prog.c`.

```
1 | gcc mon_Prog.c -o exe      #Permet de générer un executable "exe"  
2 | ./exe                    #Permet de lancer l'exécutable
```

Remarque

On se référera au chapitre sur le paradigme de la programmation modulaire pour une pratique de compilation plus évoluée.

1.3 Les expressions

Une expression est constituée des lexèmes vus précédemment.

1.3.1 Identificateur

Une notation d'affectation peut commencer par:

- Une lettre (min,MAJ)
- Un underscore (`_`)

et peut contenir des entiers.

⚠ Elle ne peut pas commencer par un entier. Ex: `"5Id_Min"` n'est pas valide.

1.4 Types de bases

Le C est un langage à typage statique. En conséquence de quoi, il faut faire figurer le type de chacun des objets ; de la sorte, lors de la compilation, un espace mémoire suffisant est alloué à chaque variable.

1.4.1 Type entier

Les valeurs entières sont caractérisées par le mot-clé **int**. Toutefois, il est possible de spécifier l'amplitude numérique et le signe de la valeur entière à stocker. Pour ce faire, il faut faire précéder **int** par **short** / **long** / **long long**, et éventuellement de **signed** / **unsigned** si l'entier est positif ou bien de signe quelconque.

	Encodage (bits)	Valeur minimal	Valeur maximale
char	8	-127	127
short	16	-32 767	32 767
int	32	-32 767	32 767
unsigned int	32	0	32 767
long	32	-2 147 483 647	2 147 483 647
long long	64	-2^{64}	$2^{64} - 1$

Table 2: Domaines & encodages du type **int**

1.4.2 Type float

Les valeurs réelles sont typées par le mot-clé **float**. Elles sont représentées syntaxiquement comme suit: **float** **x**=15.8;

	Encodage (bits)	Valeur minimal	Valeur maximale
float	32	0	32 767
double	64	-2 147 483 647	2 147 483 647
long double	128	-2^{64}	$2^{64} - 1$

Table 3: Domaines & encodages du type **int**

Sur un ordinateur, on utilise les nombres à virgule flottante de la forme $x = s * m * b^e$ où b est la base ; $s \in -1, +1$ est le signe ; la mantisse m , ou significande, précise les chiffres significatifs ; l'exposant e donne l'ordre de grandeur. Pour plus de détails, se référer à l'annexe.

1.4.3 Type bool

Il n'existe pas de type booléen à proprement parlé en C. En effet, on assimile la valeur **vrai** à 1 et **faux** à 0. De fait, il est possible de manipuler des valeurs booléennes en opérant sur $\{0; 1\}$ ou bien en définissant des *macros*. Cette notion de macros n'ayant pas encore été introduite ; il sera préférable (et équivalent) d'utiliser la librairie `stdbool.h` qui définit les constantes **true** **false** ainsi que type **bool**.

Remarque

On se référera à la sous-section Calcul propositionnel page 19 (Annexe) concernant les rappels en **logique mathématiques**.

1.4.4 Type caractère

Le type caractère est désigné par le mot-clé **char**, quoique les caractères puissent être assimilés à des entiers selon leur code ASCII.

Les **char** sont codés sur un octet (8 bits) et tolèrent les opérations arithmétiques propres aux entiers. De plus, les objets de ce type sont explicitement représentés entre guillemets simples: **'a'**

```
1 | char caractere='c';
2 | printf("Le caractère suivant dans la table ASCII est: %c",caractere+1);
3 | // Affiche 'd'
```

	dec		dec		dec
	32	@	64	'	96
!	33	A	65	a	97
"	34	B	66	b	98
#	35	C	67	c	99
\$	36	D	68	d	100
%	37	E	69	e	101
&	38	F	70	f	102
'	39	G	71	g	103
(40	H	72	h	104
)	41	I	73	i	105
*	42	J	74	j	106
+	43	K	75	k	107
,	44	L	76	l	108
-	45	M	77	m	109
.	46	N	78	n	110
/	47	O	79	o	111

	dec		dec		dec
0	48	P	80	p	112
1	49	Q	81	q	113
2	50	R	82	r	114
3	51	S	83	s	115
4	52	T	84	t	116
5	53	U	85	u	117
6	54	V	86	v	118
7	55	W	87	w	119
8	56	X	88	x	120
9	57	Y	89	y	121
:	58	Z	90	z	122
;	59	[91	{	123
<	60		92		124
=	61]	93	}	125
>	62	^	94	-	126
?	63	_	95	DEL	127

Table 4: Code ASCII

1.4.4.1 Manipulation des caractères **A faire**

1.4.5 Type chaîne de caractère

Les chaînes de caractères sont de type **char***. Lorsqu'elles sont définies explicitement, elles sont implémentées entre guillemets doubles: **"ainsi"**.

Remarque

En réalité, les chaînes de caractères sont des tableaux de caractères. On se référera à la sous-section XX page XX pour plus de détails sur les chaînes de caractères.

1.5 Les opérateurs

1.5.1 Opérateur d'affectation

L'affectation d'une expression e à variable x de type t s'effectue avec l'opérateur $=$.
De plus, lors de la compilation de l'instruction:

```
1 | t1 x = e;
```

l'expression e est convertie dans le type de la variable, en l'occurrence $t1$ ici. On appelle cela le *transtypage* (*cast* en anglais).

Afin de présenter les différents opérateurs, on utilisera la notation suivante:

$$\bullet \left|_F^n \quad \text{où} \begin{cases} n & \text{est l'arité de l'opérateur} \\ S & \text{est le type du résultat} \end{cases}$$

1.5.2 Opérateurs arithmétiques

Le type de retour est $\text{type} = \begin{cases} \text{int} & \text{si les opérandes sont de types int} \\ \text{float} & \text{si tout ou partie des opérandes sont de type float} \end{cases}$

Addition	Soustraction	Multiplication	Division entière	Modulo	Division réelle
$+ \left _{type}^2$	$- \left _{type}^2$	$* \left _{type}^2$	$/ \left _{type}^2$	$\% \left _{int}^2$	$/ \left _{float}^2$

1.5.3 Opérateurs de comparaison

$$< \left|_{bool}^2 \quad <= \left|_{bool}^2 \quad > \left|_{bool}^2 \quad >= \left|_{bool}^2 \quad != \left|_{bool}^2 \quad == \left|_{bool}^2$$

⚠ Test d'égalité sur les float

On ne réalise pas de test d'égalité sur les flottant en utilisant l'opérateur `==`

Lorsque l'on souhaite tester l'égalité entre deux réels, il faut réaliser un encadrement à ε près, ε étant la précision souhaitée.

Remarque

└ L'oubli du double égal, au profit d'un `=` est l'une des principales source d'erreur de compilation.

1.5.4 Opérateurs booléens

$$\&\& \left|_{bool}^2 \quad || \left|_{bool}^2 \quad ! \left|_{bool}^1$$

Une expression booléenne est évaluée de gauche à droite par le compilateur; en conséquence de quoi, l'évaluation cesse dès qu'il est possible de déterminer le résultat. C'est ce que l'on appelle **l'évaluation paresseuse**. De fait, on prendra garde à renseigner des expressions logiques judicieuses

à des fins d'efficacité et de gestion d'erreur (*segmentation fault* par exemple).

```
1  int i=0;
2  // Tableau de taille N
3  int T[N];
4
5  /*      c1          c2      */
6  if( (i<N) && (T[i]==1)){
7      /* Corps de boucle */
8  }
```

Exemple:

Ici, la clause c_1 ne pourrait être intervertie avec la clause c_2 , sous peine d'effectuer un dépassement de la mémoire en exécutant le test $T[i]==1$ avec $i > N$. De fait, dès que $i \geq N$, la clause c_2 n'est pas évaluée.

1.5.5 Compléments

1.5.5.1 Opérateurs composés

Il est possible d'antéposer un opérateur arithmétique à l'opérateur d'affectation pour opérer en ayant pour opérande la variable d'affectation et l'expression à affecter. Dès lors si $*$ est un opérateur, alors `type x*= expression;` \longleftrightarrow `type x=x*expression;`

1.5.5.2 Opérateurs d'incrément

Pour incrémenter une variable (lui ajouter 1), respectivement décrémenter, il est possible d'utiliser les opérateurs `++`_{int}¹ et resp. `--`_{int}¹. Généralement, on utilise cet opérateur de façon postfixée.

1.5.5.3 Opérateur de coercition

En C, les objets sont munis d'un type explicite. En conséquence de quoi, il existe un opérateur (*ici en utilisation préfixe*) permettant de changer le type des objets, lorsqu'une conversion s'avère nécessaire.

$$(type) \Big|_{\text{long unsigned int}}^1$$

1.6 Instructions et structures de contrôle

1.6.1 Les variables

Une variable est un triplet (**type**, **identificateur**, **valeur**) où l'**identificateur** suit les conventions d'écriture mentionnée dans la sous-section subsection 1.3.1, le **type** détermine l'espace mémoire nécessaire à allouer pour stocker la **valeur** en question.

1.6.1.1 Déclaration

On déclare une ou plusieurs variables de même type comme suit:

```
1  int var1, var2;
2  float var3;
```

Et même instancier ces variables dès leurs déclarations:

```
1  // Rappels:
2  char var3='l'; // Un caractère
3  char* var4="prenom"; // Une chaîne de caractère
```

1.6.1.2 Portée

Il existe deux catégories de variables.

- Les variables temporaires: Placée dans la **pile**, ces variables sont automatiquement enregistrées en mémoire lors de la compilation, et supprimées dès qu'elles n'ont plus vocation à servir; typiquement, lors de la terminaison d'une fonction secondaire dans laquelle elles seraient déclarées.
- Les variables permanentes: Elles se voient allouer un espace mémoire dans le **segment de données** invariant pendant la durée d'exécution de tout le programme. Ces variables sont précédées lors de leur déclaration par le mot-clé **static** et sont initialisées à 0.

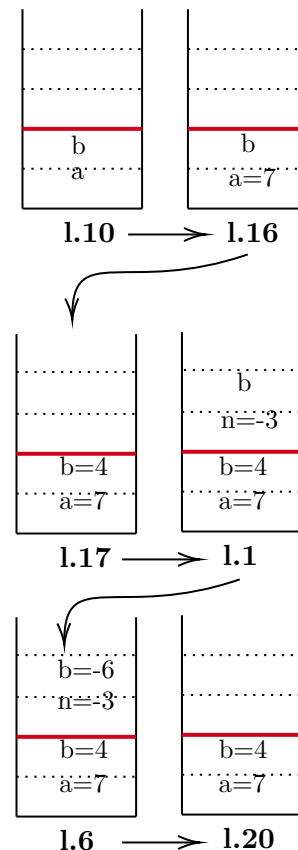
De plus, on distingue les variables **locales**, lorsqu'elles sont définies dans le corps d'une fonction; et les variables **globales**, déclarées en dehors de toute fonction dans le fichier source .c.

- Les variables locales n'ont de valeur, et d'existence que dans le corps de la fonction ou elles sont déclarées. En effet, lors de l'appel à la fonction, le processeur alloue à chacune des variables déclarées un espace mémoire dans la **pile**. Une fois l'exécution de la fonction achevée, l'ensemble des variables déclarées sont supprimées et l'espace dans la pile libéré.
- Les variables globales sont des variables permanentes. Celles-ci n'interfèrent pas avec les variables locales qui auraient le même identificateur.

1.6.1.3 Évolution de la pile

Il est important de comprendre le fonctionnement de la mémoire. Considérons un programme simple, pour lequel nous décrirons l'évolution de la pile au fur et à mesure de l'exécution du programme.

```
1 void afficheNombre(int n){
2 // Affiche le nombre n et son double
3 // Préconditions: -
4
5     int b;
6     b=2*n;
7     printf("Le nombre est %d, son
8     ↪ double est %d",n,b);
9 }
10
11 int main(){
12 // Déclaration des variables
13 int a;
14 int b;
15
16 // Initialisation des variables
17 scanf("%d",&a); // Saisie de a
18 ↪ par l'utilisateur
19 scanf("%d",&b); // Saisie de b
20
21 afficheNombre(b-a);
22 return 0;
23 }
```



Explications

Lors de l'exécution d'une fonction, avant même la lecture des instructions, le processeur alloue un espace mémoire pour chacune des variables déclarées. Par ailleurs, les arguments des fonctions sont **passés par valeurs**. Ce faisant, une copie locale de ceux-ci sont effectués. Ensuite, lignes par

lignes, les différentes instructions sont traitées, avec entre autre les différentes initialisations. Lorsque l'exécution de la fonction secondaire s'achève, les variables sont dépilées.

1.6.2 Les fonctions

Une fonction est une factorisation du programme, qui peut-être exploitée dans l'ensemble de celui-ci.

1.6.2.1 Définition

La définition d'une fonction suit la syntaxe suivante:

```
1 | type identificateur(type arg1,..., type argn){  
2 |  
3 |     [declaration de variables locales]  
4 |  
5 |     corps de la fonction  
6 |  
7 | }
```

- `type` est le type de retour de la fonction. En conséquence de quoi, le corps de la fonction fait mention d'une ou plusieurs instructions `return expression;` où `expression` est de type `type`.
- `identificateur` suit les règles de construction mentionnée dans la sous-section [Identificateur](#)
- Est encapsulée l'énumération des arguments, ici appelés **paramètres formels**, précédés de leurs types. Si la fonction ne prend pas de paramètres, on écrira `()` ou `(void)`

De plus, une fonction peut s'appeler au sein d'elle-même, c'est ce que l'on appelle la récursivité; et faire appel à d'autres fonctions. Toutefois, à l'instar du OCaml par exemple, il est impossible de définir des fonctions auxiliaires dans le corps même d'une fonction.

Tout programme contient au moins une fonction, en C, c'est la fonction `main` qui à le prototype suivant: `int main(int argc, char* argv)` ou dans des cas d'utilisation plus basique `int main(void)`.

1.6.2.2 Déclaration

Toutefois, pour que le compilateur soit au fait de l'existence de telle ou telle fonction, il est nécessaire qu'avant chaque appel, elle ait été déclarée. Pour se faire, il faut mentionner le prototype de chacune des fonctions du programme avant tout appel à celle-ci. La déclaration suit la syntaxe suivante:

```
1 | type identificateur(type_1 arg1,..., type_n argn);
```

Remarque

On se référera à la section XXX pour comprendre quand et où la déclaration des fonctions s'avère nécessaire.

1.6.2.3 Appel

L'appel à une fonction suit la syntaxe suivante:

```
identificateur(arg1,...,argn);
```

Les arguments donnés à la fonction constituent les **paramètres effectifs** et doivent concorder avec la définition de la fonction.

Les paramètres effectifs sont **passés par valeurs** lors de l'exécution de la fonction. Ce faisant, le compilateur réalise une copie locale à la fonction des arguments et les ajoute dans pile. Par conséquent, il est important de comprendre qu'en fournissant seulement la valeur des arguments, les fonctions ne sont pas en mesure de les modifier à l'extérieur de celle-ci. On verra par la suite pour qu'opérer de façon effective sur des variables extérieure à la fonction, il est nécessaire de **passer par référence** les arguments. (XXX)

1.6.2.4 Compléments

→ La fonction main

→ Fonction à paramètres variables

1.6.3 Structures conditionnelles

1.6.3.1 L'alternative

L'alternative est une structure de contrôle conditionnelles en ce qu'elle permet l'exécution d'un certain nombre d'instructions si et seulement si une expression booléenne est vérifiée.

```
1  if(cond){
2      // Corps du if
3  }
4  else{
5      // Corps du else
6  }
```

Remarque

Il n'est pas nécessaire qu'il y ait un **else**.

1.6.3.2 Opérateur ternaire

condition ? expression : expression

L'opérateur conditionnel (?:) requiert trois opérandes

→ une condition qui est une expression logique convertie en **bool**

→ deux expressions résultantes

Si la condition est évaluée à **true**, alors la première expression est évaluée. Sinon, la seconde est évaluée.

Exemple

```
1  int main(){
2      int i;
3      scanf("%d",&i); // Saisie de i par l'utilisateur
4  }
```

```

5      int j=i%2?i/2:(i+1)/2;
6
7      return 0;
8  }
```

Ici l'opérateur % correspond à l'opérateur de la congruence. En l'occurrence, si l'utilisateur à saisi une valeur paire pour i, alors j apprend $\frac{i}{2}$ sinon $\frac{i+1}{2}$. (Ici, $j = \lceil \frac{i}{2} \rceil$)

Remarque

Cet opérateur est peu utilisé compte-tenu de son manque de lisibilité.

1.6.3.3 Le branchement multiple

L'objectif du branchement multiple est de s'épargner une répétition de `if(expression==x)` ou x admet des valeurs dans un ensemble fini de taille relativement faible et défini explicitement. Ce faisant, il s'agit de "matcher" l'expression avec ses différentes possibilités. Dans le cas où l'expression ne correspond à aucun des cas énumérés, alors ce sont les instructions contenues dans `default` qui sont exécutés.

```

1  switch(expression){
2      case exp:
3          // Cas 1
4      break;
5      ...
6      case exp:
7          // Cas n
8      break;
9      default:
10         // Cas de défaut
11     break;
12 }
```

Remarque

Il n'est pas nécessaire de faire figurer une alternative `default`.
Le mot-clé `break` permet de sortir du dernier corps {}.

1.6.4 Les boucles

1.6.4.1 Tant que...

```

1  while(condition){
2      // Corps de boucle
3  }
```

1.6.4.2 Pour ...

```
1 | for(valeur initiale;valeur finale;incrémentation){  
2 |     // Corps de boucle  
3 | }
```

1.6.4.3 Faire ... Tant que

```
1 | do{  
2 |     // Corps de boucle  
3 | }while(condition);
```

Fonctionnement similaire à une boucle while, à la différence qu'au moins un tour de boucle est réalisé.

1.6.4.4 Passes-droit

Il est possible de mettre un terme à l'exécution d'une boucle, et en cas de boucle imbriquée de sortir de la boucle la plus interne, à l'aide du mot-clé: **break**;

Étant donnée une étiquette, *i.e.* un mot-clé inscrit à un endroit du code, d'effectuer un saut dans le programme à l'endroit indiqué par l'étiquette par le biais de l'instruction **goto** étiquette.

Une telle instruction **goto** est à proscrire.



2 Types composés

2.1 Les structures

Une structure est une collection finie d'objets de types quelconques, indexable. On accède aux éléments de la structure, appelés champs, par le biais de leur identificateur. La déclaration d'une structure est équivalente à la déclaration d'un nouveau type.

2.1.1 Définition

La définition d'une structure suit la syntaxe suivante:

```
1 struct identificateur{
2     type_1 id_1; // Champ 1 de type 1
3     ...
4     type_n id_n; // Champ n de type n
5 };
```

⚠ On veillera à ne pas oublier le ; à la fin de la déclaration afin d'éviter une erreur de compilation. De plus, une structure ne peut pas comporter plus de cent vingt-sept membres.

2.1.2 Instanciation

Pour instancier une structure précédemment définie:

<code>struct identificateur nom;</code>	→ déclaration
<code>struct identificateur nom={v_1,...,v_n};</code>	→ initialisation séquentielle
<code>nom.champ_1=valeur;</code>	→ initialisation de champ
<code>nom.champ_1</code>	→ accès la valeur d'un champ

Remarque

Il ne faut pas confondre la déclaration comme *définition* d'une structure ; et la déclaration d'une *instanciation* de celle-ci.

2.2 Les énumérations

2.3 Les unions

2.4 Les tableaux

Un tableau est une structure de donnée contenant des éléments homogènes (*i.e.* de même type), indexable.

2.4.1 Déclaration

En C, les tableaux sont des objets statiques, et possèdent donc une taille fixée lors de leur déclaration.

⚠ Afin d'accéder aux éléments, ou de parcourir le tableau, il est primordial de connaître la taille de celui-ci afin de ne pas provoquer d'erreur de segmentation en tentant d'accéder à une zone mémoire non attribuée. En conséquence de quoi, **une déclaration de tableau est toujours précédée par l'initialisation d'une variable indiquant sa taille.**

2.4.1.1 Déclaration seule

```
type identificateur[taille];
```

→ **type** est le type des éléments du tableau → **taille** est le nombre d'éléments que peut contenir au plus le tableau **identificateur**

2.4.1.2 Déclaration et initialisation


Lorsque les éléments du tableau sont à valeur dans un ensemble explicite de taille raisonnable, il est possible d'initialiser celui-ci dès la déclaration:

```
type identificateur[N]={e1,...,eN};
```

Remarque

Il est possible de procéder à des initialisations sélectives, ainsi que d'omettre la taille du tableau en laissant le soin au compilateur de déterminer celle-ci en fonction de la position du dernier éléments. Toutefois, cela ne semble pas être des pratiques de programmation judicieuses.

Si on souhaite copier un tableau `tab2` préalablement défini et initialisé. L'instruction `tab1=tab2` n'agira pas comme on le souhaite. En réalité, **les tableaux sont des adresses mémoire sur la première valeur stockée du dit tableau**. Ainsi, lors de la précédente instruction, vous avez simplement créé une autre variable permettant d'accéder au **même tableau**.

 Il est impossible de copier un tableau à partir d'un autre en réalisant seulement une affectation. Pour ce faire, il faut copier un à un les éléments.

```
1  for (i = 0; i < N; i++){
2      tab1[i] = tab2[i];
3  }
```

2.4.2 Accès aux valeurs

Les tableaux sont indicés par $\llbracket 0; N-1 \rrbracket$ où N est la taille du tableau. On accède à la $i^{\text{ème}}$ $\in \llbracket 0; N-1 \rrbracket$ case du tableau avec l'instruction:

```
identificateur[i]
```

2.4.3 Tableaux multidimensionnels

Sur le même principes que les tableaux à une dimensions, les tableaux multidimensionnels se déclarent comme suit:

```
type identificateur[N] [M] ... [P];
```

En particulier, en dimension 2, une déclaration et initialisation suivrait la syntaxe suivante:

```
1  type tableau[N] [M]={e11,...,e1M}    // N paires d'accolades
2                                  {e21,...,e2M}    // à M éléments
3                                  ...
4                                  {eN1,...,eNM}
5                                  }
```

3 La mémoire

3.1 Les pointeurs

3.2 Le cas des tableaux

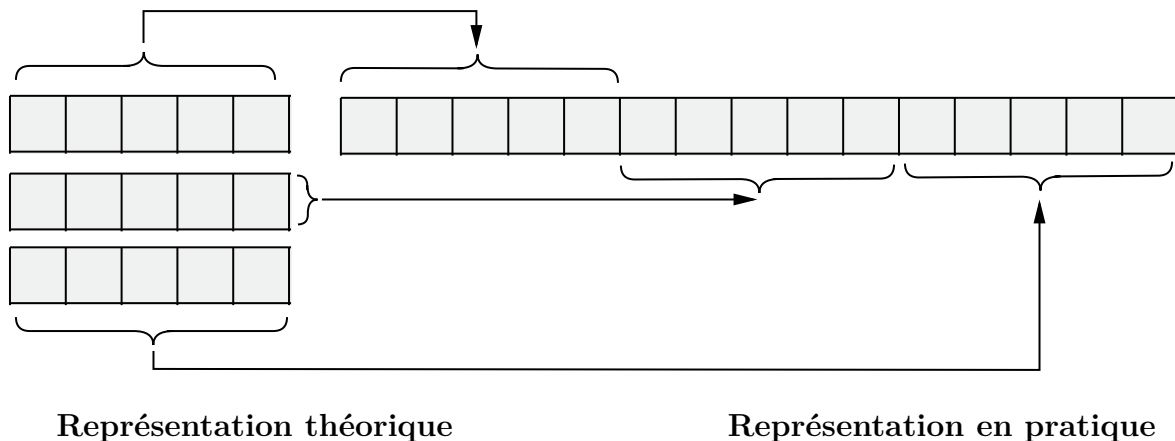
`tab + 1` est implicitement convertie en `tab + sizeof(int)`

`expression[indice]` Ce qui est équivalent à l'expression suivante.

`*(expression + indice)`

3.2.1 Tableaux multidimensionnels

Toutefois, en pratique, les tableaux multidimensionnels sont linéarisés et enregistrés en mémoire comme un tableau 1D selon la convention *row-major order*, autrement dit lignes par lignes.



De ce fait, si on considère un tableau $\mathcal{T} = (t_{i,j})$ de taille $N \times M$, alors:

$$\forall (i, j) \in [0..N[\times [0..M[, \begin{cases} \&t_{i,j} = j + N * i & \text{adresse de la case} \\ \mathcal{T}[i][j] & \text{accès à la valeur en C} \end{cases}$$

3.3 Le cas des structures

portée des arguments, pointeur sur structure, esp mem des champs pas forcément contiguë,

4 Annexe

4.1 Boîte à outils

4.1.1 Test ici

4.1.1.1 Encore un autre

Les types simples			
Type	Exemple	Spécificateur de format	Encodage
int	5	%d	
float	3.2	%f	
char	'a'	%c	
char*	"ici"	%s	
bool	true false	%d	
unsigned	1520	%u	
long unsigned	15697853	%lu	
Les fonctions d'entrée sortie			

4.2 Représentation des réels

4.3 Calcul propositionnel

On définit $\mathbb{B} = \{V; F\}$ (i.e l'ensemble des valeurs booléennes), ainsi que les fonctions suivantes:

$$+ = \begin{pmatrix} \mathbb{B} \times \mathbb{B} & \rightarrow & \mathbb{B} \\ (V, V) & \mapsto & V \\ (V, F) & \mapsto & V \\ (F, V) & \mapsto & V \\ (F, F) & \mapsto & F \end{pmatrix} \quad \times = \begin{pmatrix} \mathbb{B} \times \mathbb{B} & \rightarrow & \mathbb{B} \\ (V, V) & \mapsto & V \\ (V, F) & \mapsto & F \\ (F, V) & \mapsto & F \\ (F, F) & \mapsto & F \end{pmatrix} \quad \bar{\bullet} = \begin{pmatrix} \mathbb{B} & \rightarrow & \mathbb{B} \\ V & \mapsto & F \\ F & \mapsto & V \end{pmatrix}$$

Ce faisant, $(\mathbb{B}, +, \times, \bar{\bullet})$ est appelé **Algèbre de Boole**.

Remarque

$\bar{\bullet}$ est un opérateur unaire semblable à la conjugaison dans \mathbb{C} . De plus, $+, \times, \bar{\bullet}$ sont des lois internes sur la sémantique des formules propositionnelles. On les distingue des opérateur \vee, \wedge, \neg qui font sens syntaxiquement sur $\mathbb{F}_p(\mathcal{Q})$ l'ensemble des formules propositionnelles construit par induction à partir d'un ensemble de symboles \mathcal{Q} , appelés **variables propositionnelles**. On se référera à un cours de logique pour aller plus loin.

Propriété Soit $(\mathbb{B}, +, \times, \bar{\bullet})$, alors on a les propriétés suivantes:

- $+$ et \times sont associatives, commutatives et distributives l'une par rapport à l'autre
- $+$ admet pour élément neutre **F**
- \times admet pour élément neutre **V**
- **V** est absorbant pour $+$
- **F** est absorbant pour \times
- $\bar{\bullet}$ est involutive

Démonstrations: En utilisant les définitions ou en réalisant des tables de vérité.

Propriété Soit $a, b \in \mathbb{B}$,

$$\overline{(a + b)} = \bar{a} \times \bar{b} \text{ et } \overline{(a \times b)} = \bar{a} + \bar{b}$$

Remarque

L'implémentation en C de $+, \times, \bar{\bullet}$ correspond à $\&\&, ||, !$

Liste de Code de programme

References

- [1] Jean-Jacques Lévy. “Informatique fondamentale - Ecole Polytechnique”. **in()**.
- [2] Luc Maranget. “Cours de compilation - Ecole Polytechnique”. **in**(2004): **pages** 49–50.