

Langage C Semestre 2

F. Loup
Yverdon-les-Bains
Janvier 2003

INFORMATION
TECHNOLOGY



INDEX

1	INTRODUCTION	6
1.1	Présentation du langage C	6
1.1.1	Historique	6
1.1.2	Caractéristiques	6
1.1.3	Structure d'un programme C	7
1.1.4	Structure d'une instruction	7
1.1.5	Structure d'un bloc	7
1.1.6	Structure d'une fonction	8
1.1.7	Règles d'écriture des programmes C	8
1.1.8	Exemple de programme	9
2	LEÇON 1 : CONSTANTES, VARIABLES, TYPES DE BASE ET OPERATEURS ARITHMETIQUES	11
2.1	Objectifs de la leçon	11
2.2	Description	11
2.2.1	Identificateurs	11
2.2.2	Types de base	12
2.2.3	Caractère	12
2.2.4	Entier :	12
2.2.5	Les réels :	12
2.3	Tableau récapitulatif des différents types	13
2.4	Les constantes	13
2.4.1	Constantes entières :	13
2.4.2	Constantes réelles :	14
2.4.3	Constantes caractères :	14
2.4.4	Constantes chaînes de caractères :	15
2.5	Définition d'une variable simple	16
2.6	Les opérateurs arithmétiques	16
2.7	Exercice	17
2.7.1	Calculs	17
2.7.2	En-tête de programme	17
3	LEÇON 2 : STRUCTURER VOS IDEES	18
3.1	Objectifs de la leçon	18
3.2	Structogrammes	18
3.2.1	Le programme, la fonction, la procédure	18
3.2.2	La séquence	18
3.2.3	L'alternative	19
3.2.4	La boucle	19
3.2.5	Assemblage	20
3.2.6	Exemple	20
3.3	Exercices	21
3.3.1	Structogramme 1	21
3.3.2	Structogramme 2	21
4	LEÇON 3 : STRUCTURES DE CONTROLE	22
4.1	Objectifs de la leçon	22



4.2	Structures de contrôle	22
4.2.1	L'instruction if	22
4.2.2	L'instruction while	23
4.2.3	L'instruction do ... while	24
4.2.4	L'instruction for	25
4.2.5	L'instruction switch	26
4.2.6	L'instruction break	28
4.2.7	L'instruction continue	28
4.3	Exercice	28
4.3.1	Itérations	28
5	LEÇON 4 : OPERATEUR D'AFFECTATION, INCREMENTATION, DECREMENTATION ET OPERATEUR DE SEQUENCE.....	29
5.1	Objectifs de la leçon	29
5.2	Les opérateurs d'affectation	29
5.2.1	Valeur à gauche (lvalue) :	29
5.2.2	L'opérateur d'affectation :	29
5.2.3	Affectation combinée :	29
5.3	Incrémentation / décrémentation	30
5.3.1	Pré-incrémentation/pré-décrémentation :	30
5.3.2	Post-incrémentation/post-décrémentation :	30
5.4	L'opérateur de séquence	30
5.5	Exercice	31
5.5.1	Opérateur d'affectation, incrémentation, décrémentation et opérateur de séquence 31	
6	LEÇON 5 : OPERATEURS RELATIONNELS ET EXPRESSIONS BOOLEENNES 32	
6.1	Objectifs de la leçon	32
6.2	Opérateurs relationnels	32
6.3	Les expressions booléennes.....	32
6.3.1	Généralités :	32
6.3.2	Opérateurs booléens :	33
6.4	Exercices.....	33
6.4.1	Opérateurs relationnels, booléens.....	33
6.4.2	Calcul de la somme et de la moyenne	34
7	LEÇON 6 : AFFICHAGE ET SAISIES.....	35
7.1	Objectifs de la leçon	35
7.2	La fonction printf.....	35
7.3	La fonction scanf	38
7.4	La macro getchar	40
7.5	La fonction gets	41
7.6	Exercices.....	42
8	LEÇON 7 : DEFINITION ET UTILISATION D'UNE VARIABLE TABLEAU	43
8.1	Objectifs de la leçon	43
8.2	Définition d'une variable tableau.....	43
8.3	Utilisation d'une variable tableau.....	44
8.4	Exercices.....	45
8.4.1	Gestion de données sous forme d'un tableau	45



9	LEÇON 8 : CLASSE DE STOCKAGE, MODIFICATEURS ET INITIALISATION DES VARIABLES	46
9.1	Objectifs de la leçon	46
9.2	Classes de stockage et modificateurs.....	46
9.2.1	Classe de stockage d'une variable :	46
9.2.2	Variables globales :	46
9.2.3	Variables locales :.....	46
9.2.4	Modificateurs de type :.....	48
9.3	Initialisation des variables	48
9.3.1	Initialisation explicite	48
9.3.2	Initialisation implicite.....	48
9.4	Exercices.....	48
10	LEÇON 9 : CONVERSIONS DE TYPES	50
10.1	Objectifs de la leçon	50
10.2	Conversions	50
10.2.1	Conversions implicites :	50
10.2.2	Conversions explicites (casting) :.....	51
10.3	Exercice	52
10.3.1	Analyse de texte.....	52
11	LEÇON 10 : LES POINTEURS.....	53
11.1	Objectifs de la leçon	53
11.2	Les pointeurs.....	53
11.2.1	Définition d'une variable pointeur	53
11.2.2	Opérateur d'adresse.....	54
11.2.3	Opérateur d'indirection	54
11.3	Arithmétique des pointeurs.....	55
11.3.1	Incrémement/décrémement de pointeur :	55
11.3.2	Addition/soustraction de 2 pointeurs :	55
11.3.3	Comparaison de pointeurs :	55
11.3.4	Affectation de pointeurs :	55
11.4	Pointeurs et tableaux.....	56
11.5	Les tableaux et les chaînes.....	57
11.6	Exercices.....	58
11.6.1	Les pointeurs.....	58
12	LEÇON 11 : LES FONCTIONS	59
12.1	Objectifs de la leçon	59
12.2	Les fonctions.....	59
12.2.1	Généralités	59
12.2.2	Passage des paramètres.....	59
12.2.3	L'appel d'une fonction.....	60
12.2.4	Définition ANSI d'une fonction.....	60
12.2.5	Déclaration ANSI d'une fonction	61
12.2.6	Les prototypes.....	61
12.2.7	Passage par "référence"	62
12.2.8	L'instruction return	63
12.2.9	Arguments de main()	63
12.3	Exercice	65



12.3.1	Programmation efficace à l'aide des fonctions.....	65
13	LEÇON 12 : OPERATEURS DE MANIPULATION DE BITS ET, STRUCTURES	66
13.1	Objectifs de la leçon	66
13.2	Opérateurs de manipulation de bits	66
13.3	Les structures.....	67
13.3.1	Déclaration de type structure :.....	68
13.3.2	Utilisation des champs d'une structure :	69
13.4	Exercice	71
13.4.1	Manipulations de bits.....	71
14	PROJETS.....	72
14.1	Graphique d'une fonction mathématique	72
14.2	Calcul du prix du transport	72
14.3	Cercle.....	73
14.4	Carnet d'adresses.....	74
14.5	Gestion des I/O d'un PC.....	74
14.6	Gestion des feux d'un carrefour	75



1 Introduction

Ce cours est destiné à donner des bases concernant la programmation structurée en C et C++. Ce fascicule ne contient pas toutes les informations concernant ces langages mais, une introduction aux principales fonctions qui les caractérisent. Dans le cas où l'étudiant voudrait ce perfectionner dans un de ces langages, un grand nombre de livres traitent de ces sujets et, la plupart des compilateurs C/C++ contiennent une aide qui vous donnera toutes les informations nécessaires.

1.1 Présentation du langage C

1.1.1 Historique

- Le langage C est né en 1972 dans les laboratoires de la Bell Telephone (AT&T) des travaux de Brian Kernighan et Dennis Ritchie.
- Il a été conçu à l'origine pour l'écriture du système d'exploitation UNIX (90-95% du noyau est écrit en C) et s'est vite imposé comme le langage de programmation sous UNIX.
- Très inspiré des langages BCPL (Martin Richard) et B (Ken Thompson), il se présente comme un "super-assembleur" ou "assembleur portable". En fait c'est un compromis entre un langage de haut niveau (Pascal, Ada ...) et un langage de bas niveau (assembleur).
- Il a été normalisé en 1989 par le comité X3J11 de l'American National Standards Institute (ANSI).

1.1.2 Caractéristiques

- **Langage structuré**, conçu pour traiter les tâches d'un programme en les mettant dans des blocs.
- Il produit des **programmes efficaces** : il possède les mêmes possibilités de contrôle de la machine que l'assembleur et il génère un **code compact et rapide**.
- **Déclaratif** : normalement, tout objet C doit être déclaré avant d'être utilisé. S'il ne l'est pas, il est considéré comme étant du type entier.
- **Format libre** : la mise en page des divers composants d'un programme est totalement libre. Cette possibilité doit être exploitée pour rendre les programmes lisibles.
- **Modulaire** : une application pourra être découpée en modules qui pourront être compilés séparément. Un ensemble de programmes déjà opérationnels pourra être réuni dans une librairie. Cette aptitude permet au langage C de se développer de lui-même.
- **Souple et permissivité** : peu de vérifications et d'interdits, hormis la syntaxe. Il est important de remarquer que la tentation est grande d'utiliser cette caractéristique pour écrire le plus souvent des atrocités.
- **Transportable** : les entrées/sorties sont réunies dans une librairie externe au langage.



1.1.3 Structure d'un programme C

Un programme C est composé de :

Directives du pré processeur : elles permettent d'effectuer des manipulations sur le texte du programme source avant la compilation :

- inclusion de fichiers
- substitutions
- macros
- compilation conditionnelle

Une directive du pré processeur est une ligne de programme source commençant par le caractère dièse (#).

Le pré processeur (*/lib/cpp*) est appelé automatiquement, en tout premier, par la commande */bin/cc*.

Déclarations/définitions :

Déclaration : la déclaration d'un objet C donne simplement ses caractéristiques au compilateur et ne génère aucun code.

Définition : la définition d'un objet C déclare cet objet et crée effectivement cet objet.

Fonctions : Ce sont des sous-programmes dont les instructions vont définir un traitement sur des variables.

Des commentaires : éliminés par le pré processeur, ce sont des textes compris entre */** et **/*.

On ne doit pas les imbriquer et ils peuvent apparaître en tout point d'un programme (sauf dans une constante de type chaîne de caractères ou caractère).

Pour ignorer une partie de programme il est préférable d'utiliser une directive du pré processeur (**#if 0 ... #endif**)

1.1.4 Structure d'une instruction

Une instruction élémentaire est une **expression terminée par le caractère ;** (point virgule)

L'instruction nulle est composée d'un ; seul.

Il est recommandé, afin de faciliter la lecture et le "debugage" de ne mettre qu'une seule instruction par ligne dans la source du programme.

1.1.5 Structure d'un bloc

- Un bloc est une suite d'instructions élémentaires délimitées par des accolades { et }
- Un bloc peut contenir un ou plusieurs blocs inclus
- Un bloc peut commencer par des déclarations/définitions d'objets qui seront locaux à ce bloc. Ces objets ne pourront être utilisés que dans ce bloc et les éventuels blocs inclus à ce bloc.



1.1.6 Structure d'une fonction

Une fonction est un bloc de code d'une ou plusieurs instructions qui peuvent renvoyer une valeur à l'expression qui l'utilise.

- Elle peut retourner une valeur à la fonction appelante.
- Le programme principal est une fonction dont le nom doit impérativement être **main**.
- Les fonctions ne peuvent pas être imbriquées.

La forme générale (en ANSI) d'une fonction est :

```
[classe] [type] nom( [liste_de_parametres_formels] )  
bloc_de_la_fonction
```

Les éléments entre [] dans cette syntaxe, signifie que cet élément est facultatif, car une valeur par défaut existe.

1.1.7 Règles d'écriture des programmes C

Afin d'écrire des programmes C lisibles, il est important de respecter un certain nombre de règles de présentation :

- Ne jamais placer plusieurs instructions sur une même ligne
- Utiliser des identificateurs significatifs
- Grâce à l'indentation des lignes, on fera ressortir la structure syntaxique du programme (tabulation).
- Les valeurs de décalage les plus utilisées sont de 2, 4 ou 8 espaces.
- On laissera une ligne blanche entre la dernière ligne des déclarations et la première ligne des instructions.
- Une accolade fermante est seule sur une ligne (à l'exception de l'accolade fermante du bloc de la structure `do ... while`) et fait référence, par sa position horizontale, au début du bloc qu'elle ferme.
- Aérer les lignes de programme en entourant par exemple les opérateurs avec des espaces.
- Il est nécessaire de commenter les listings. Eviter les commentaires triviaux du style `/* si */` pour une instruction `'if'`.



1.1.8 Exemple de programme

```
/*
/*
/* Fichier : exp.C
/* Auteur : F. Loup
/* Date de creation : 30.10.98
/* Derniere modification : 18.01.2000
/* Historique: voir le fichier MODIF.TXT
/* Version : 2.15
/*
/* Description : Programme de démonstration
/*
/*****
#include <stdio.h> /* directives au pre processeur */
#define DEBUT -10
#define FIN 10
#define MSG "Programme de démonstration\n"

int carre(int x); /* déclaration des fonctions */
int cube(int x);

/*****
main()
/*
/* Programme principal
/*
/* F.Loup 30/10/98
/*****
{
    /* début du bloc de la fonction main */
    int i; /* définition des variables locales */
    printf(MSG);
    for ( i = DEBUT; i <= FIN ; i++ )
    {
        printf("%d carré: %d cube: %d\n", i
                , carre(i)
                , cube(i) );
    } /* fin du bloc for */
    return 0;
} /* FIN DU BLOC DE LA FONCTION MAIN */

/*****
int cube(int x)
/*
/* Fonction cube
/*
/* F.Loup 30/10/98
/*****
{
    return x * carre(x);
}
```



```
}  
  
/*****/  
int carre(int x)  
/*                                     */  
/* Fonction carre                       */  
/*                                     */  
/* F.Loup 30/10/98                       */  
/*****/  
{  
    return x * x;  
}
```



2 Leçon 1 : Constantes, variables, types de base et opérateurs arithmétiques

2.1 Objectifs de la leçon

Au terme de la leçon, l'élève doit être capable de :

- Décrire les notions de constantes et variables ainsi que leurs différences
- Utiliser les opérateurs arithmétiques du C ANSI

2.2 Description

- Constante :
Identificateur possédant une valeur invariable
- Variable :
Identificateur possédant une valeur pouvant être modifiée à tout moment

2.2.1 Identificateurs

Les identificateurs nomment les objets C (fonctions, variables...)

- C'est une suite de lettres ou de chiffres, dont le premier caractère est obligatoirement une lettre.
- Le caractère _ (souligné) est considéré comme une lettre.
- Le C distingue les minuscules des majuscules. Exemple : carlu Carlu CarLu CARLU sont des identificateurs valides et tous différents.
- La longueur de l'identificateur dépend de l'implémentation. La norme ANSI prévoit qu'au moins les 31 premiers caractères sont significatifs pour le compilateur. L'éditeur de liens peut limiter le nombre de caractères significatifs des identificateurs à un nombre plus petit.
- Un identificateur ne peut pas être un mot réservé du langage :

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

Recommandations :

- Utiliser des identificateurs significatifs.
- Réserver l'usage des identificateurs en majuscules pour le pré processeur.
- Réserver l'usage des identificateurs commençant par le caractère _ (souligné) à l'usage du compilateur.



2.2.2 Types de base

C'est à partir de ces types de base que l'on peut construire tous les autres types, dits types dérivés (tableaux, structures, unions...).

2.2.3 Caractère

Le type **char** représente un caractère parmi l'ensemble des caractères du jeu de caractères de la machine.

Le code des caractères utilisés n'est pas défini par le langage. Dans la grande majorité des cas c'est le code ASCII.

- Taille : 1 octet (8 bits)
- intervalle : -128 à +127 ou 0 à 255 suivant les machines

Il peut être :

- **signed char** : intervalle de -128 à 127.
- **unsigned char** : intervalle de 0 à 255.

Ces 2 derniers types rendent les programmes plus portables, en forçant la plage de valeur.

2.2.4 Entier :

Le type **int** permet de représenter les nombres entiers.

Il correspond à un mot machine. Sa taille dépend donc de la taille du processeur utilisé. Elle est en général de 16 ou 32 bits.

Il peut être qualifié par :

- **short** ou **long** : spécifie la taille de l'emplacement mémoire utilisé.
- **unsigned** : entier positif
- **signed** : entier signé (par défaut)

int est facultatif dès qu'il est précédé de **short**, **long** ou **unsigned**.

La seule chose que précise la norme ANSI est que :

taille(short) <= taille(int) <= taille(long)

2.2.5 Les réels :

Ils permettent de représenter un nombre en virgule flottante.

Ils se divisent en :

- **float** : pour les nombres en virgule flottante simple précision
- **double** : pour les nombres en virgule flottante double précision
- **long double** : pour les nombres en virgule flottante à précision étendue.

Les fichiers d'en-tête **limits.h** et **float.h** contiennent des constantes symboliques qui précisent les tailles et les valeurs limites des entiers et des flottants.



2.3 Tableau récapitulatif des différents types

Type	Bits	Min.	Max.
char	8	-128 ou 0	127 ou 255
signed char	8	-128	127
unsigned char	8	0	255
short	16	-32768	32767
unsigned short	16	0	65535
int	16 ou 32	-32768 ou -2147483648	32768 ou 2147483647
unsigned (int)	16 ou 32	0	65535 ou 4294967295
long	32	-2147483648	2147483647
unsigned long	32	0	4294967295
float	32	3.4 E-38	3.4 E+38
double	64	1.7 E-308	1.7 E+308
long double	128	3.4 E-4932	3.4 E+4932

2.4 Les constantes

Chaque constante a une valeur et un type.

Les expressions constantes sont évaluées à la compilation; ainsi l'expression constante $1 + 2 * 3$ est interprétée comme l'expression entière 7.

2.4.1 Constantes entières :

Les constantes entières peuvent être exprimées sous les formes suivantes :

- **Décimale** : exemple : 1234
- **Octale** : le premier chiffre est obligatoirement un zéro. Exemple : 0177
- **Hexadécimale** : le nombre commence par **0x** ou **0X**.
On peut utiliser les lettres minuscules ou majuscules pour les chiffres hexadécimaux.
Exemple : 0x1Bf 0XF2a

Les entiers trop grands pour tenir dans un int seront considérés comme du type long. On peut imposer à une constante entière d'être du type :

- **long** : en la suffixant **l** ou **L** à la fin (il est préférable d'utiliser le L majuscule afin d'éviter la confusion entre le l minuscule et le chiffre 1).
Exemple : 123L
- **unsigned** : en la suffixant **u** ou **U** à la fin.
Exemple : 3456U , 78UL (unsigned long)



2.4.2 Constantes réelles :

Les constantes réelles sont, par défaut, de type **double**.

Elles peuvent être exprimées sous les formes suivantes :

- **Décimale** : exemple : 1234.56 4. -3456
- **Scientifique** : exemple : 1.23e-12 12.3456E4

Elles peuvent aussi inclure un suffixe :

- **f** ou **F** : impose le type float à la constante. Exemple : 1.23f
- **l** ou **L** : impose le type long double. Exemple : 123.4567E45L

2.4.3 Constantes caractères :

Une constante de type caractère est écrite entre apostrophes. Elles peuvent être exprimées sous les formes suivantes :

- **Normale** : 'a'
- **Octale** : '\007'
- **Hexadécimale** : '\x1b'
- **Symbolique** :

NOTATION	CODE ASCII (hexa)	SIGNIFICATION
'\n'	0x0A	saut de ligne
'\t'	0x09	tabulation horizontale
'\v'	0x0B	tabulation verticale
'\a'	0x07	beep
'\b'	0x08	retour arrière
'\r'	0x0D	retour chariot
'\f'	0x0C	saut de page
'\\'	0x5C	backslash
'\"'	0x2C	simple quote
'\"'	0x22	double quote
'\?'	0x3F	point d'interrogation

La valeur d'une constante caractère est égale à la valeur numérique du caractère dans le code caractère de la machine.



ASCII Codes															
Char	Hex	Oct	Dec	Char	Hex	Oct	Dec	Char	Hex	Oct	Dec	Char	Hex	Oct	Dec
Ctrl-@ NUL	00	000	0	Space	20	040	32	@	40	100	64	`	60	140	96
Ctrl-A SOH	01	001	1	!	21	041	33	A	41	101	65	a	61	141	97
Ctrl-B STX	02	002	2	"	22	042	34	B	42	102	66	b	62	142	98
Ctrl-C ETX	03	003	3	#	23	043	35	C	43	103	67	c	63	143	99
Ctrl-D EOT	04	004	4	\$	24	044	36	D	44	104	68	d	64	144	100
Ctrl-E ENQ	05	005	5	%	25	045	37	E	45	105	69	e	65	145	101
Ctrl-F ACK	06	006	6	&	26	046	38	F	46	106	70	f	66	146	102
Ctrl-G BEL	07	007	7	'	27	047	39	G	47	107	71	g	67	147	103
Ctrl-H BS	08	010	8	(28	050	40	H	48	110	72	h	68	150	104
Ctrl-I HT	09	011	9)	29	051	41	I	49	111	73	i	69	151	105
Ctrl-J LF	0A	012	10	*	2A	052	42	J	4A	112	74	j	6A	152	106
Ctrl-K VT	0B	013	11	+	2B	053	43	K	4B	113	75	k	6B	153	107
Ctrl-L FF	0C	014	12	,	2C	054	44	L	4C	114	76	l	6C	154	108
Ctrl-M CR	0D	015	13	-	2D	055	45	M	4D	115	77	m	6D	155	109
Ctrl-N SO	0E	016	14	.	2E	056	46	N	4E	116	78	n	6E	156	110
Ctrl-O SI	0F	017	15	/	2F	057	47	O	4F	117	79	o	6F	157	111
Ctrl-P DLE	10	020	16	0	30	060	48	P	50	120	80	p	70	160	112
Ctrl-Q DC1	11	021	17	1	31	061	49	Q	51	121	81	q	71	161	113
Ctrl-R DC2	12	022	18	2	32	062	50	R	52	122	82	r	72	162	114
Ctrl-S DC3	13	023	19	3	33	063	51	S	53	123	83	s	73	163	115
Ctrl-T DC4	14	024	20	4	34	064	52	T	54	124	84	t	74	164	116
Ctrl-U NAK	15	025	21	5	35	065	53	U	55	125	85	u	75	165	117
Ctrl-V SYN	16	026	22	6	36	066	54	V	56	126	86	v	76	166	118
Ctrl-W ETB	17	027	23	7	37	067	55	W	57	127	87	w	77	167	119
Ctrl-X CAN	18	030	24	8	38	070	56	X	58	130	88	x	78	170	120
Ctrl-Y EM	19	031	25	9	39	071	57	Y	59	131	89	y	79	171	121
Ctrl-Z SUB	1A	032	26	:	3A	072	58	Z	5A	132	90	z	7A	172	122
Ctrl-[ESC	1B	033	27	;	3B	073	59	[5B	133	91	{	7B	173	123
Ctrl-\ FS	1C	034	28	<	3C	074	60	\	5C	134	92		7C	174	124
Ctrl-] GS	1D	035	29	=	3D	075	61]	5D	135	93	}	7D	175	125
Ctrl-^ RS	1E	036	30	>	3E	076	62	^	5E	136	94	~	7E	176	126
Ctrl_ US	1F	037	31	?	3F	077	63	_	5F	137	95	DEL	7F	177	127

2.4.4 Constantes chaînes de caractères :

Elles sont constituées d'une séquence de caractères, et délimitées par le caractère " (guillemet).

Toutes les notations de caractères (normale, octale, hexadécimale ou symbolique) sont utilisables dans les chaînes.



Exemples :

```
"Le langage C"  
"Bonjour\n"  
"\\"Hello\\""  
"Anti-slash : \\" ""
```

- Un caractère nul ('\0') est ajouté automatiquement à la fin de la chaîne.
- Les constantes chaînes de caractères sont stockées en zone de mémoire permanente.
- Elles sont considérées comme des tableaux de caractères dont la dimension est celle de chaîne plus 1 (pour le caractère nul placé à la fin du tableau).
- Une chaîne peut tenir sur plusieurs lignes :

Exemples :

```
"Ceci est une constante chaine\  
sur 2 lignes"  
ou  
"Ceci est une autre forme"  
"de chaine sur 2 lignes"
```

- Il n'existe pas d'opérateurs sur les chaînes de caractères, mais il existe une librairie de fonctions de manipulations de chaînes qui permet d'effectuer toutes les opérations courantes.

2.5 Définition d'une variable simple

Les variables doivent obligatoirement être déclarées avant d'être utilisées.

La syntaxe générale d'une définition ou déclaration de variable est de la forme :

```
classe modificateur type identificateur ;
```

- **classe** : (facultative) elle détermine le mode de stockage, la durée de vie et la visibilité de la variable.
Elle peut prendre une des valeurs suivantes :
auto extern register static
- **modificateur** : (facultatif) il peut prendre une des valeurs suivantes :
const volatile
- **type** : c'est un type de base ou un type utilisateur.

Exemple :

```
int x; /* x est un entier */  
float x1, x2; /* x1 et x2 sont des entiers */  
static double delta; /* delta est un double de classe static */  
extern unsigned char rep; /* rep est un char non signé externe */
```

2.6 Les opérateurs arithmétiques

Tous les opérateurs qui suivent acceptent des opérandes pouvant être des entiers, des caractères ou des réels (sauf l'opérateur modulo).

opérateur	opération
-----------	-----------



-	MOINS UNAIRE
+	plus unaire
*	multiplication
/	division
+	addition
-	soustraction
%	modulo : reste de la division entière (entre entiers)

Remarques :

- **% (modulo)** : reste de la division entière (entre entiers seulement). Si un des opérandes est négatif, le signe du reste dépend de l'implémentation. En général il est de même signe que le dividende.

Exemple :

```
printf("%d\n", -3 % 2 ); /* affichage de -1 */  
printf("%d\n", 3 % -2 ); /* affichage de +1 */
```

- Il n'existe pas d'opérateur d'exponentiation.

2.7 Exercice

2.7.1 Calculs

Soient a, b et c des variables entières auxquelles ont été affectés les valeurs 8, 3 et -5. Déterminer la valeurs des expressions arithmétiques suivantes :

- a) $a + b + c$
- b) $2 * b + 3 (a - c)$
- c) a / b
- d) $a \% b$
- e) a / c
- f) $a \% c$
- g) $a * b / c$
- h) $a * (b / c)$
- i) $(a * c) \% b$
- j) $a * (c \% b)$

2.7.2 En-tête de programme

Créer un fichier xxx.cpp contenant une en-tête en se basant sur l'exemple de la page 9. Ce fichier servira de base à chaque nouveau programme.



3 Leçon 2 : Structurer vos idées

3.1 Objectifs de la leçon

Au terme de cette leçon, l'élève doit être capable de :

- Tracer un structogramme à partir d'une description d'un problème à résoudre.

3.2 Structogrammes

Afin de structurer ses idées avant de taper des lignes de codes et, d'obtenir le code le plus efficace possible, on utilise une représentation graphique du fonctionnement final du programme ou, d'une partie de celui-ci.

Plusieurs formes de base sont à disposition afin de créer une représentation fidèle à ces idées.

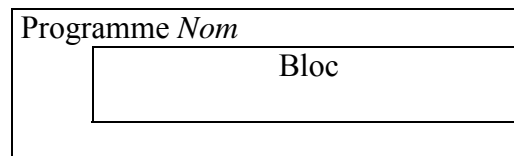
Attention :

La lecture d'un structogramme s'effectue toujours de haut en bas, il n'est pas possible de sortir sur un des côtés de cette suite logique.

3.2.1 Le programme, la fonction, la procédure

Ce bloc permet de délimiter un programme, une fonction ou une procédure.

Représentation graphique :



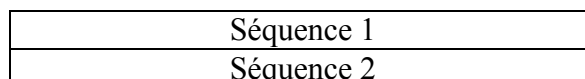
3.2.2 La séquence

La séquence permet d'attribuer une valeur à une variable numérique ou alphanumérique, ou d'effectuer une opération de lecture ou d'écriture.

Exemples :

- Chiffre1 = 121
- Ecrire sur l'écran la valeur de la variable chiffre1
- Lire le clavier

Représentation graphique :



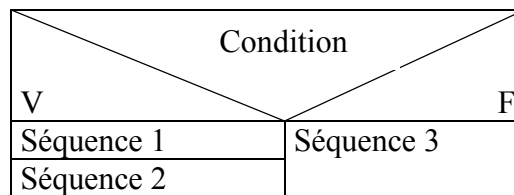


3.2.3 L'alternative

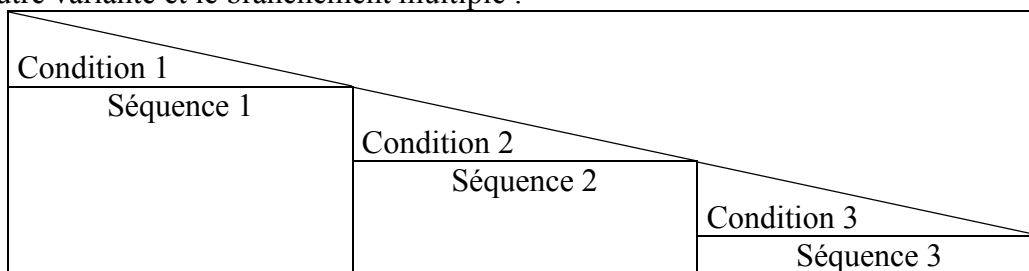
L'alternative permet de faire le choix entre deux exécutions possibles, soit les instructions dépendants du test vérifié vrai ou faux.

Exemple :

Si température_extérieure < 22
Allumer chauffage
Sinon
Ne rien faire
Fin du test

Représentation graphique :

Une autre variante et le branchement multiple :



3.2.4 La boucle

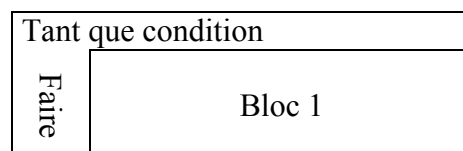
La boucle permet d'exécuter autant de fois une séquence d'instructions qu'une condition le demande.

Exemple :

Tant que nbre_de_feuille < 15
Prendre une feuille supplémentaire
Compter les feuilles
Fin de la boucle

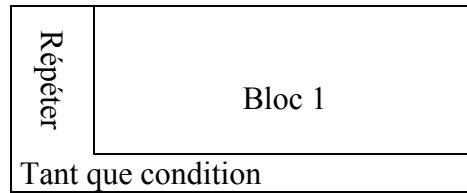
Représentation graphique :

Variante 1 :

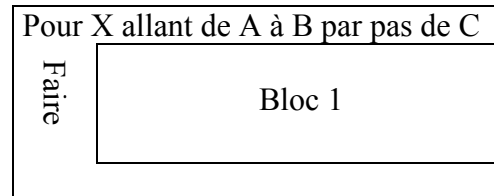




Variante 2 :



Variante 3 :

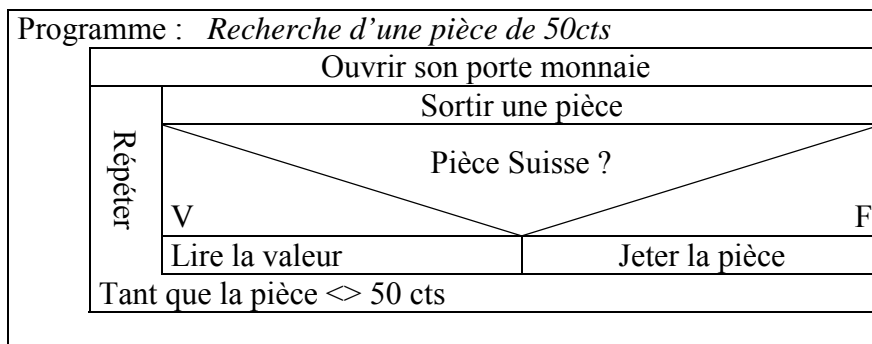


3.2.5 Assemblage

Afin de créer une représentation graphique fidèle aux idées que l'on a, il est nécessaire d'assembler ces différents blocs graphiques. Les possibilités suivantes sont utilisables :

- Appondre de haut en bas des blocs graphiques
- Incorporer un bloc à la place d'une instruction ou séquence
- Faire appel à un autre structogramme dans une instruction, afin de rendre le structogramme plus petit et plus lisible

3.2.6 Exemple





3.3 Exercices

3.3.1 Structogramme 1

Tracer le structogramme GNS du programme qui accepte, en entrée, **trois réels en ordre croissant**. Un quatrième nombre, quelconque, est saisi et, doit être positionné par rapport aux trois nombres classés. Seule les trois nombres de valeurs les plus faibles doivent être affichées à l'écran.

Exemple :

Exemple	Entrée des trois premiers nombres en ordre croissant	Entré du quatrième nombre	Sortie
1	A=5 B=7 C=9	D=4	A=4 B=5 C=7
2	A=5 B=7 C=9	D=6	A=5 B=6 C=7
3	A=5 B=7 C=9	D=7	A=5 B=7 C=7
4	A=5 B=7 C=9	D=10	A=5 B=7 C=9

3.3.2 Structogramme 2

On saisit deux nombres plus grand que zéro, A et B, au clavier. Après la saisie, on doit garantir que le nombre A soit plus petit que le nombre B. Si ce n'est pas le cas alors, on devra permuter les deux nombres. On termine par l'affichage de A et B après la vérification.

4 Leçon 3 : Structures de contrôle

4.1 Objectifs de la leçon

Au terme de la leçon, l'élève doit être capable de :

- Décrire le fonctionnement des structures de contrôle
- Utiliser les structures de contrôle afin de créer des opérations fonctionnelles

4.2 Structures de contrôle

4.2.1 L'instruction if

Syntaxe :

```

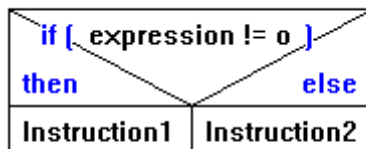
if ( expression )
    instruction1 ou bloc d'instruction1
[else
    instruction2 ou bloc d'instruction2]
    
```

Description :

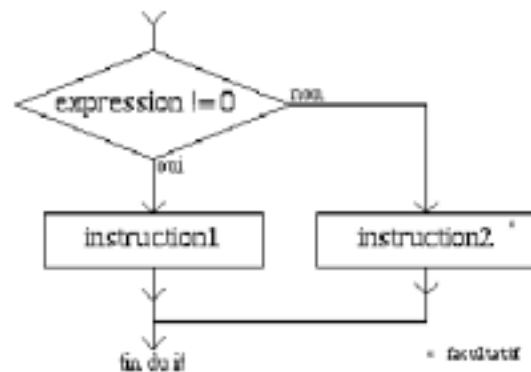
La valeur de l'expression est évaluée et, si elle est non nulle, instruction1 est exécutée sinon c'est l'instruction 2 qui est exécutée (si elle existe). L'instruction 1 et l'instruction 2 peuvent être des instructions simples ou des blocs. La clause else peut être omise et se rapporte toujours au dernier if visible.

Diagramme syntaxique du if :

Structogramme GNS



Organigramme



Exemples:

```

if ( i < 10 )
    i++;
    
```

```

if ( delta > 0 )
    printf("2 racines reelles\n");
else
    if ( delta == 0 )
        printf("racine double\n");
    else
    {
        printf("pas de racines reelles\n");
    }
    
```



```
    printf("entrer une nouvelle valeur\n");  
  }  
  
  if (a) /* equivalent : if ( a != 0 ) */  
    i++;  
  
  if (an % 4 == 0 && an % 100 != 0 || an % 400 == 0)  
    printf("annee bissextile\n");
```

4.2.2 L'instruction while

Cette instruction permet de répéter une instruction (ou un bloc) tant qu'une condition est vraie.

Syntaxe :

```
while ( expression )  
  instruction ou bloc d'instructions
```

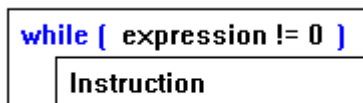
Description :

L'instruction est exécutée de façon répétitive aussi longtemps que le résultat de l'expression est non nul.

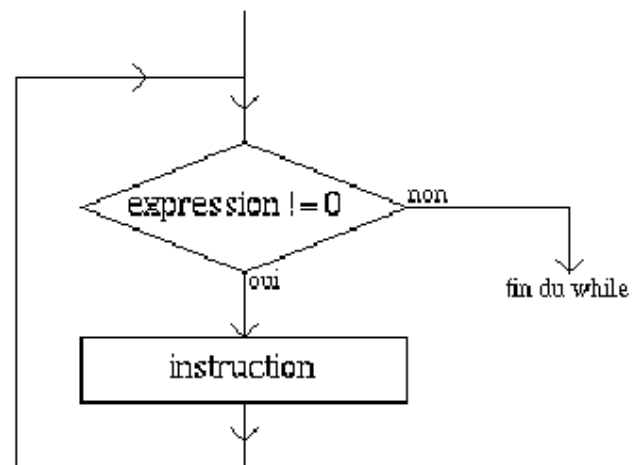
L'expression est évaluée avant chaque exécution de l'instruction.

Diagramme syntaxique :

Structogramme GNS



Organigramme



Exemple :

```
int i = 0 ;  
/* affiche les nombres de 0 - 9 */  
while (i != 10)  
{  
    printf("%d ", i);  
    i++;  
}
```



4.2.3 L'instruction do ... while

Syntaxe :

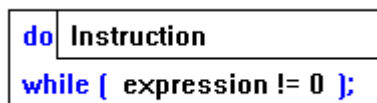
```
do
    instruction ou bloc d'instructions
while ( expression != 0 );
```

Description :

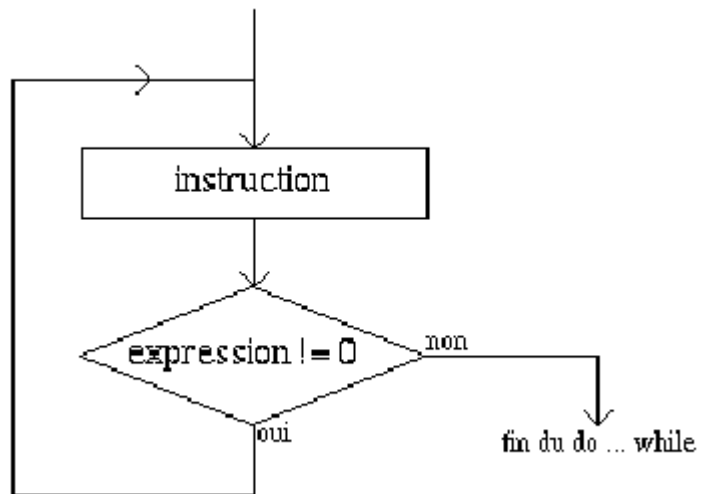
Cette instruction est similaire à la précédente, le test a lieu après chaque exécution de l'instruction, de fait l'instruction est au moins exécutée une fois.

Diagramme syntaxique :

Structogramme GNS



Organigramme



Exemples :

```
int i = 1 , somme = 0;
/* somme des nombres de 1 a 10 */
do
{
    somme += i;
    ++i;
} while (i <= 10);
```

```
char c;
/* test de reponse */
do
{
    printf("Voulez-vous continuer (o/n) ? ");
    c = getchar();
} while(c != 'o' && c != 'n')
```



4.2.4 L'instruction for

Syntaxe :

```
for ( [expression1] ; [expression2] ; [expression3] )  
    instruction ou bloc d'instructions
```

Description :

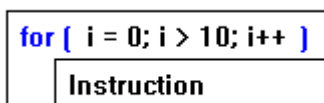
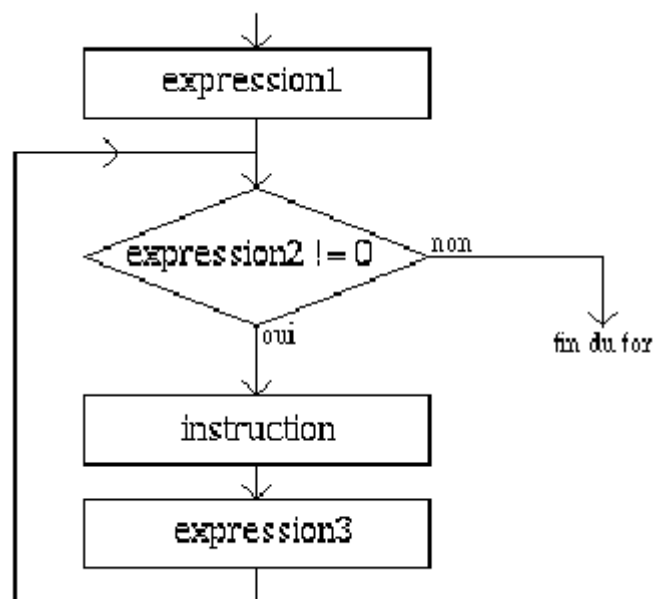
L'instruction est répétée tant que la valeur de expression2 est non nulle.

Avant la première itération, expression1 est évaluée;
En général elle sert à initialiser les variables de la boucle.

Après chaque itération de la boucle, expression3 est évaluée.
En général elle sert à incrémenter le compteur de la boucle.

La boucle for est équivalente à la structure suivante :

```
expression1;  
while ( expression2 )  
{  
    instruction ou bloc d'instruction;  
    expression3;  
}
```

Diagramme syntaxique :Structogramme GNSOrganigramme**Remarques :**

- Toutes les expressions sont facultatives. Si expression2 n'existe pas, elle sera supposée vraie (valeur 1).
- L'opérateur , peut être utilisé dans expression1 et expression3.

**Exemples :**

```
for (i=0; i<10; i++)
    somme += tab[i];

for( ; ; )
    ; /* boucle infinie ne faisant rien ! */

for( i = 0 , j = 0 ; i < 10 ; i++ , j++ )
    ...
```

4.2.5 L'instruction switch

L'instruction switch permet un choix multiple en fonction de l'évaluation d'une expression.

Syntaxe :

```
switch ( expression )
{
    case e1 : instruction1 ...
    case e2 : instruction2 ...
    ...
    case e3 : instruction3 ...
    default : instruction_default ...
}
```

Description :

L'évaluation de expression doit donner pour résultat une valeur de type int.

e1, e2, e3 ... sont des expressions constantes qui doivent être un entier unique de type int ou char.

La valeur de expression est recherchée successivement parmi les valeurs des différentes expressions constantes e1, e2, e3.

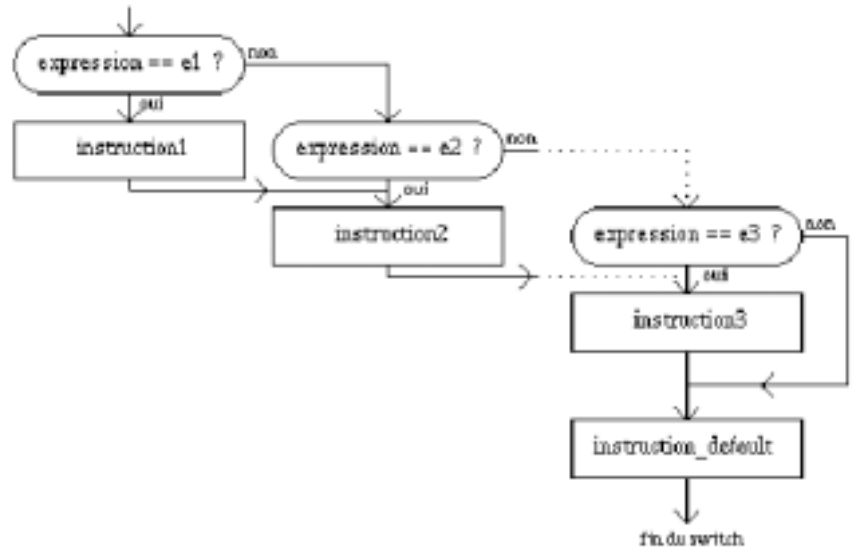
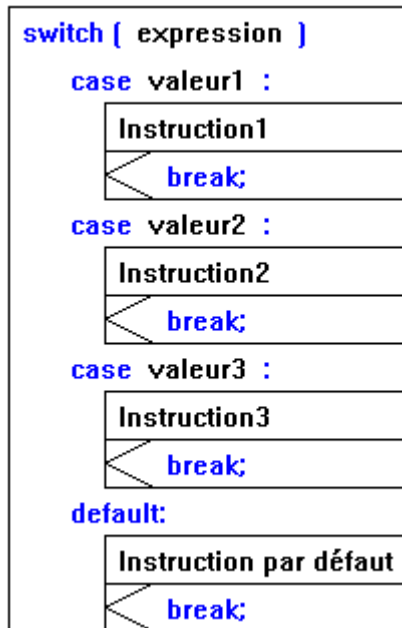
En cas d'égalité les instructions (facultatives) correspondantes sont exécutées jusqu'à une instruction break ou jusqu'à la fin du bloc du switch (et ceci indépendamment des autres conditions case).

S'il n'y a pas de valeur correspondante, on exécute les instructions du cas default (s'il existe).

Diagramme syntaxique :

Structogramme GNS

Organigramme

**Exemple :**

```
switch (car)
{
    case 'a' :
    case 'A' :
    case 'e' :
    case 'E' :
        v++;
        break;
    case ' ' :
        espace++;
        break;
    default : c++;
}
```

Remarque :

Il est possible de mettre plusieurs valeurs dans le même test.

Exemple :

```
switch (car)
{
    case 'a', 'A' :
    case 'e', 'E' :
        v++;
        break;
    case ' ' :
        espace++;
        break;
    default : c++;
}
```



4.2.6 L'instruction **break**

L'instruction **break** provoque le passage à l'instruction qui suit immédiatement le corps de la boucle *while*, *do-while*, *for* ou *switch*.

4.2.7 L'instruction **continue**

L'instruction **continue** fait passer à l'itération suivante les instructions *while*, *do-while* ou *for*.

Elle est équivalente à un "goto" à la fin du bloc.

Exemple :

```
for (i = -10; i <= 10; i++)
{
    if (i == 0)
        continue;           // pour éviter la division par zéro
    tab[i] = 1 / i;
}
```

Pour la boucle *for*, l'instruction **continue** fait passer à l'évaluation de *expr3* (*i++*).

4.3 Exercice

4.3.1 Itérations

Créer un programme qui accepte une variable comprise entre 1 et 10.

Si cette valeurs est inférieure à 1, corriger la à 1 et, si elle est supérieur à 10, corriger la à 10. Et afficher à l'écran un message indiquant l'erreur.

Ensuite, utiliser une boucle pour savoir combien de fois on doit multiplier cette valeur par deux pour que le résultat soit supérieur ou égal à 100.

Travail à effectuer :

- Structogramme GNS du programme
- Créer le code C/C++ à partir du structogramme, utiliser l'instruction

```
cprintf("Valeur inférieur à 1\n");
cprintf("Valeur supérieur à 10\n");
cprintf("Résultat = %d\n", val);
```

pour afficher le résultat à l'écran.



5 Leçon 4 : Opérateur d'affectation, incrémentation, décrémentement et opérateur de séquence

5.1 Objectifs de la leçon

Au terme de la leçon, l'élève doit être capable de :

- Décrire une lvalue
- Affecter une valeur à une lvalue
- Incrémenter ou décrémenter la valeur d'une variable
- Reconnaître l'opérateur de séquence dans un programme et comprendre son fonctionnement

5.2 Les opérateurs d'affectation

5.2.1 Valeur à gauche (lvalue) :

Une valeur à gauche (lvalue comme left value) est une expression qui peut apparaître à gauche de l'opérateur d'affectation.

Une lvalue possède une adresse en mémoire.

Exemples :

```
int alpha , beta ;
char tab[10];
5                /* la constante 5 n'est pas une lvalue */
alpha           /* alpha est une lvalue */
alpha + beta    /* alpha + beta n'est pas une lvalue */
tab[1]         /* tab[1] est une lvalue */
tab            /* n'est pas une lvalue. C'est un
pointeur constant */
```

5.2.2 L'opérateur d'affectation :

En C, l'affectation est un opérateur comme les autres. L'objet à gauche du (égal) se voit affecter (après conversion éventuelle) la valeur retournée par l'expression de droite.

On peut donc écrire :

```
alpha = beta = 0;
```

qui est équivalent à : `alpha = (beta = 0);` car l'opérateur = est associatif de droite à gauche. La valeur 0 est affectée à beta puis la valeur retournée par cette affectation est mise dans alpha.

5.2.3 Affectation combinée :

Il existe en C un ensemble d'opérateurs d'affectations.

Syntaxe :

```
lvalue op= expression
```



où op est un des opérateurs suivants :

* / % + - << >> & ^ |

A op= B est équivalent à : A = A op B.

Exemple :

x += 3; /* équivalent à : x = x + 3 */

Un seul accès à la variable à gauche de l'opérateur est effectué.

5.3 Incrémentation / décrémentation

On doit souvent incrémenter ou décrémentation une variable. Le langage C offre 2 opérateurs (unaires) pour effectuer ces opérations :

- ++ : incrémentation de 1
- -- : décrémentation de 1

5.3.1 Pré-incrémentation/pré-décrémentation :

Les opérateurs d'incrément/décrément sont placés devant la variable. L'incrément/décrément est effectuée puis l'utilisation de la variable est faite.

Exemple : l'instruction : **v1 = ++v2;** est équivalente à :

```
v2 = v2 + 1;  
v1 = v2;
```

La valeur de l'expression ++v2 est la valeur de v2 après incrémentation.

5.3.2 Post-incrémentation/post-décrémentation :

Les opérateurs d'incrément/décrément sont placés après la variable. L'utilisation de la variable est effectuée avant l'incrément/décrément.

Exemple : l'instruction : **v1 = v2++;** est équivalente à :

```
v1 = v2;  
v2 = v2 + 1;
```

La valeur de l'expression v2++ est la valeur de v2 avant incrémentation.

5.4 L'opérateur de séquence

L'opérateur , (virgule) permet d'évaluer les différentes expressions dans l'ordre.

La valeur retournée est la valeur de la dernière expression évaluée.

Exemple :



```
a = ( b=10 , b+20 );  
/* equivalent : b = 10; a = b + 20; */  
temp = a , a = b , b = temp;
```

5.5 Exercice

5.5.1 Opérateur d'affectation, incrémentation, décrémentation et opérateur de séquence

Soit a et b des variables entières auxquelles ont été affectés les valeurs 8 et 3.
Déterminer la valeurs des expressions arithmétiques suivantes :

- | | | |
|----------------------|-----|-----|
| a) a *= b | a = | |
| b) b += 5 | b = | |
| c) a = ++b | a = | b = |
| d) a = b++ | a = | b = |
| e) a /= --b | a = | b = |
| f) a %= b | a = | |
| g) a = (b++, b + 10) | a = | b = |
| h) a /= b | a = | |
| i) a = (b = 20, b++) | a = | b = |
| j) a = 20 - (--b) | a = | b = |

Finir l'exercice 4.4.1 concernant les itérations



6 Leçon 5 : Opérateurs relationnels et expressions booléennes

6.1 Objectifs de la leçon

Au terme de la leçon, l'élève doit être capable de :

- Décrire les opérateurs relationnels
- Utiliser les opérateurs fonctionnels dans un programme fonctionnel
- Décrire une expression booléennes
- Trouver la solution d'une expression booléenne

6.2 Opérateurs relationnels

Le résultat de la comparaison entre 2 expressions vaut :

- **0** si le résultat de la comparaison est **faux**
- **1** si le résultat de la comparaison est **vrai**.

Opérateurs relationnels

Opérateur	signification
==	test si égalité
!=	test si différent
<	test si inférieur
<=	test si inférieur ou égal
>	test si supérieur
>=	test si supérieur ou égal

Exemples :

- **rep = (c > 5);**
rep est égal à 1 si c est supérieur à 5, sinon rep est égal à 0.
- **if (a == b) ...**
teste si a est égal à b.

6.3 Les expressions booléennes

6.3.1 Généralités :

Il n'existe pas en C de type booléen. La convention suivante est utilisée :

- Une expression est vraie si elle est non nulle.
- Une expression est fausse si elle est égale à zéro.

Exemples :

```
5 est une expression toujours vraie
i = 5 est une expression toujours vraie (affectation)
i == 5 est une expression vraie si i est égal à 5
```

Il ne faut pas confondre l'opérateur de test d'égalité avec l'opérateur d'affectation .
Une instruction telle que : `if (a = 5) ...` est valide et toujours vraie.



6.3.2 Opérateurs booléens :

- **&&** réalise le "ET booléen" entre 2 expressions. Retourne 1 (vrai) si les 2 expressions valent 1, 0 sinon.
- **||** réalise le "OU booléen" entre 2 expressions. Retourne 1 (vrai) si l'une ou l'autre (ou les deux), des 2 expressions valent 1, 0 sinon.
- **!** Réalise la "négation booléenne" de son unique opérande (opérateur unaire). Retourne 1 si la valeur de son opérande est 0 (faux), 0 sinon.

OP2	OP1	OP1 && OP 2	OP1 OP2	!OP1
0	0	0	0	1
0	1	0	1	0
1	0	0	1	1
1	1	1	1	0

L'évaluation des expressions booléennes s'effectue de gauche à droite (sauf pour l'opérateur {~!}) et elle est stoppée dès que le résultat de l'expression devient définitif.

Exemples :

- $a \geq 5 \ \&\& \ a \leq 10$
 - retourne 1 si a est compris entre 5 et 10.
 - du fait de la priorité plus forte des opérateurs \geq et \leq par rapport à $\&\&$, cette expression est équivalente à : $(a \geq 5) \ \&\& \ (a \leq 10)$
- $a > 5 \ || \ b == 0$
 - retourne 1 si a est supérieur à 5 ou si b est égal à 0.
 - l'expression $b == 0$ ne sera pas évaluée si a est supérieur à 5 (évaluation courte).

6.4 Exercices

6.4.1 Opérateurs relationnels, booléens

Soient a et b des variables entières auxquelles ont été affectés les valeurs 8 et 3.
Déterminer si l'expression donne la valeur Vrai ou Faux (tracer ce qui ne convient pas).

a) $a == b$	Vrai	Faux
b) $a <= b$	Vrai	Faux
c) $a != (5 + b)$	Vrai	Faux
d) $(a == b) (a > b)$	Vrai	Faux
e) $!(b >= a)$	Vrai	Faux
f) $(a) (a < b)$	Vrai	Faux
g) $(a >= b) \ \&\& \ (b <= 10)$	Vrai	Faux
h) $(a != b) \ \&\& \ !(b <= a)$	Vrai	Faux
i) $a == (b + 1 * 2)$	Vrai	Faux
j) $a == ((b + 1) * 2)$	Vrai	Faux



6.4.2 Calcul de la somme et de la moyenne

Créer un programme qui accepte des valeurs comprise entre 1 et 6 en entrée en utilisant l'instruction scanf.

Si cette valeurs n'est pas comprise entre 1 et 6, la valeur est refusée et une nouvelle valeur est demandée.

Les opérations à effectuer sont :

- Calcul du nombre de valeurs entrée
- Calcul de la somme des valeurs entrées
- Calculs de la moyenne
- Affichage de ces différentes valeurs à l'écran

Ces opérations doivent être effectuées après chaque entrée d'une valeur.

Prévoir de pouvoir quitter le programme en entrant la valeur 0.

Un structogramme doit être fourni avec le programme.



7 Leçon 6 : Affichage et saisies

7.1 Objectifs de la leçon

Au terme de la leçon, l'élève doit être capable de :

- Lire des données sur le clavier
- Ecrire des informations sur l'écran

7.2 La fonction `printf`

- **Syntaxe :**

```
#include <stdio.h>

printf( const char *format [, arg [, arg]...]);
```

- **Description :**

Elle permet l'écriture formatée sur le flux standard de sortie *stdout* (l'écran par défaut).

La chaîne de caractères format peut contenir à la fois :

1. Des caractères à afficher,
2. Des spécifications de format.
3. Des spécifications de fin de ligne

NOTATION	CODE ASCII (hexa)	SIGNIFICATION
'\n'	0x0A	saut de ligne
'\t'	0x09	tabulation horizontale
'\v'	0x0B	tabulation verticale
'\a'	0x07	beep
'\b'	0x08	retour arrière
'\r'	0x0D	retour chariot
'\f'	0x0C	saut de page

Il devra y avoir autant d'arguments à la fonction *printf* qu'il y a de spécifications de format.

- **Valeur retournée :**

le nombre d'octets effectivement écrits ou la constante *EOF* (-1) en cas d'erreur.

- **Spécificateurs de format :**

ils sont introduites par le caractère % et se terminent par le caractère de type de conversion suivant la syntaxe suivante :

% [drapeaux] [largeur] [.precision] [modificateur] type

○ **drapeaux :**

drapeaux	Signification
rien	justifié à droite et complété à gauche par des espaces
-	justifié à gauche et complété à droite par des espaces
+	les résultats commencent toujours par le signe + ou -
espace	le signe n'est affiché que pour les valeurs négatives
#	forme alternative. Si le type de conversion est : c,s,d,i,u : sans effet : un 0 sera placé devant la valeur x, X : 0x ou 0X sera placé devant la valeur e, E, f : le point décimal sera toujours affiché g, G : même chose que e ou E, mais sans supprimer les zéros à droite

- **largeur** : elle précise la nombre de caractères n qui seront affichés.

Si la valeur à afficher dépasse la taille de la fenêtre ainsi définie, C utilise quand même la place nécessaire.

largeur	Effet sur l'affichage
n	affiche n caractères, complété éventuellement par des espaces
0n	affiche n caractères, complété éventuellement à gauche par des 0
*	l'argument suivant de la liste fournit la largeur

- **précision** : elle précise pour :
 - un entier, le nombre de chiffres à afficher
 - un réel, le nombre de chiffres de la partie décimale à afficher (avec arrondi)
 - les chaînes, le nombre maximum de caractères à afficher.

précision	Effet sur l'affichage
rien	précision par défaut : d, i, o, u, x : 1 chiffre e, E, f : 6 chiffres pour la partie décimale.
.0	d, i, o, u, x : précision par défaut e, E, f : pas de point décimal
.n	n caractères au plus
*	l'argument suivant de la liste contient la précision

- **modificateur** : Il précise comment sera interprété l'argument.

Modificateur	interprétation comme
h	un entier de type short (d, i, o, u, x, X)
l	un entier de type long (d, i, o, u, x, X)
L	un réel de type long double (e, E, f, g, G)



- **type** : type de conversion de l'argument.

Type	Format de la sortie
d ou i	entier décimal signé
o	entier octal non signé
u	entier décimal non signé
x	entier hexadécimal non signé
X	entier hexadécimal non signé en majuscules
f	réel de la forme [-]dddd.ddd
e	réel de la forme [-]d.ddd e [+/-]ddd
E	comme e mais l'exposant est la lettre E
g	format e ou f suivant la précision
G	comme g mais l'exposant est la lettre E
c	caractère
s	affiche les caractères jusqu'au caractère nul '\0' ou, jusqu'à ce que la précision soit atteinte
p	pointeur

- **Exemple :**

```
#include <stdio.h>

main()
{
    int nbre = 5;
    char *chaine = "Le langage C";
    long prix = 12.0L;
    long double result = prix * nbre;

    printf("Bonjour\n");
    printf("Nombre %d prix %ld Total %9ld\n",nbre, prix,
           prix * nbre);
    printf("%s est facile\n", chaine);
    printf("%8.2Lf \n", result);
    printf("%*.*Lf \n", 8, 2, result); /* equivalent a
           %8.2Lf */
    printf("\n");           /* affichage du caractère % */
    return 0;
}

/*-- résultat de l'exécution ----
Bonjour
Nombre 5 prix 12 Total 60
Le langage C est facile
60.00
60.00
-----*/
```



- **Exemple d'utilisation des formats numériques :**

instruction C	résultat
<code>printf("%d\n",12345);</code>	12345
<code>printf("%+d\n",12345);</code>	+12345
<code>printf("%8d\n",12345);</code>	12345
<code>printf("%8.6d\n",12345);</code>	012345
<code>printf("%x\n",255);</code>	ff
<code>printf("%X\n",255);</code>	FF
<code>printf("%#x\n",255);</code>	0xff
<code>printf("%f\n",1.23456789012345);</code>	1.234568
<code>printf("%10.4f\n",1.23456789);</code>	1.2346

7.3 La fonction *scanf*

- **Syntaxe :**

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

```
int scanf( const char *format [, arg [, arg]...]);
```

- **Description :**

La fonction *scanf* permet de faire une lecture formatée du flux standard d'entrée (le clavier par défaut).

Elle lit les caractères en entrée, les interprète en concordance avec les spécifications de format décrites dans la chaîne *format*, et place les résultats dans les arguments *arg*.

Pour pouvoir retourner les valeurs ainsi saisies, les *arg* doivent être obligatoirement des pointeurs.

- **Valeur retournée :**

le nombre de valeurs convenablement introduites ou *EOF* (-1) en cas d'erreur.

- **Remarques :**

- Les informations tapées au clavier sont d'abord mémorisées dans un tampon avant d'être traitées par *scanf*.
- La chaîne de format ne doit comporter que des spécifications de format, tout autre caractère peut amener le programme à se comporter curieusement ...

- **Spécificateurs de format :**

Ils sont introduits par le caractère % (pour-cent) et se terminent par le caractère de type de conversion suivant le format suivant :

```
% [largeur] [modificateur] type
```



- **largeur** : elle précise le nombre de caractères n qui seront lus. On peut en lire moins si l'on rencontre un séparateur (espace, tabulation, retour chariot ...) ou un caractère invalide.
- **modificateur** : Il précise la taille de l'objet recevant la valeur.

Modificateur	L'objet recevant est
h	un entier de type short int (d, i, o, u, x)
l	un entier de type long int (d, i, o, u, x) un réel de type double (e, f)
L	un réel de type long double (e,f,g)

- **type** : type de l'objet pointé par arg.

Type	Type de l'objet pointé
d	signed int exprimé en décimal
o	signed int exprimé en octal
u	unsigned int exprimé en décimal
x	int (signed ou unsigned) exprimé en hexadécimal
f,e,g	Réel à virgule flottante
c	suivant la largeur : largeur non spécifiée ou égale à 1 : caractère largeur différente de 1 : une chaîne de caractères
s	une chaîne de caractères
p	pointeur exprimé en hexadécimal

- **Exemple :**

```
#include <stdio.h>

main()
{
    int i,j;
    double d;
    char tab[81];

    printf("entier: ");
    scanf("%d", &i);
    printf("2 entiers et 1 double: ");
    scanf("%d%d%lf", &i, &j, &d);
    printf("chaîne (sans espace): ");
    scanf("%80s", tab);

    /* le caractère '\0' est automatiquement ajouté à la
    fin de la chaîne tab*/

    return 0;
}
```



7.4 La macro *getchar*

- **Syntaxe :**

```
#include <stdio.h>

int getchar();
```

- **Description :**

Elle permet la lecture d'un caractère sur l'entrée standard.

- **Valeur retournée :**

cette macro retourne :

- En cas de succès : la valeur du caractère lu
- En cas d'erreur : la constante *EOF* (-1) et positionne *errno* pour indiquer l'erreur.

- **Remarques :**

- L'appel de cette macro est bien plus rapide qu'un appel à la fonction *scanf("%c",&c)*; dans la mesure où elle ne fait pas appel à l'analyse d'une chaîne de format.

- Cette macro est définie dans *stdio.h* comme :

```
define getchar() getc(stdin)
```

- La macro *getchar* utilise le même tampon que la fonction *scanf*.

- **Exemple :**

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

```
main()
```

```
{
    int rep;

    do
    {
        printf("Voulez-vous continuer ? ");
        rep = getchar();
        printf("\n");
    } while ( rep != 'N' && rep != 'n');
    printf("fin\n") ;
    return 0;
}
```

```
/*-- résultat de l'exécution -----
Voulez-vous continuer ? u
Voulez-vous continuer ? o
Voulez-vous continuer ? n
fin
```



-----*/

- **Remarques sur le résultat de l'exécution :**

- le '\n' ainsi que les autres caractères frappés restent dans le buffer du flux d'entrée standard et le prochain *getchar()* extraira le premier d'entre eux du buffer.
- une solution consiste à vider le flux d'entrée après sa lecture par la fonction de vidage du flux *stdin* :

```
#include <stdio.h>

main()
{
    int rep;

    do
    {
        printf("Voulez-vous continuer ? ");
        rep = getchar();
        fflush(stdin); /* vidage du flux stdin */
    } while ( rep != 'o' && rep != 'n');

    return 0;
}
```

- une autre solution consiste à ne pas bufferiser les lectures du tampon d'entrée standard :

```
#include <stdio.h>

main()
{
    int rep;

    system("stty raw");
    do
    {
        printf("Voulez-vous continuer ? ");
        rep = getchar();
    } while ( rep != 'o' && rep != 'n');
    system("stty cooked");

    return 0;
}
```

7.5 La fonction *gets*

- **Syntaxe :**

```
#include <stdio.h>

char *gets(char *s);
```




8 Leçon 7 : Définition et utilisation d'une variable tableau

8.1 Objectifs de la leçon

Au terme de la leçon, l'élève doit être capable de :

- Déclarer une variable tableau
- Ecrire et lire des informations dans un tableau

8.2 Définition d'une variable tableau

Un tableau est une variable composée d'une suite séquentielle d'éléments de même type.

La syntaxe générale d'une définition de variable tableau à une dimension est de la forme :

```
classe modificateur type identificateur[exp_const_entiere] ;
```

La classe, le modificateur et le type sont les mêmes que pour les variables simples.

Le nombre d'éléments du tableau est déclaré entre les crochets par une expression constante entière évaluable par le compilateur (une variable ne peut donc pas apparaître dans l'expression définissant la taille du tableau).

Exemple :

```
int tab[10];                /* tab est un tableau de 10 entiers */
static float x[2*50+2];    /* x est un tableau de 102 réels */
char ch1[20], ch2[40];     /* 2 tableaux de caractères */
double coef[ 2 * i];       /* ERREUR : expression non constante */
```

- Pour un tableau de dimension n , les indices valides vont de 0 à $n-1$
- Pour une déclaration de tableau ou lorsque le tableau est initialisé lors de sa définition, la dimension du tableau peut être omise

Exemple :

```
int tab[] = { 10, 20, 30 }; /* tableau (de 3 entiers)
                             initialisé */
extern int coef[];         /* déclaration d'un tableau */
*/
```


Mémoire, Zone tab	Adresse x	10
	Adresse x + 1	20
	Adresse x + 2	30

- Il est recommandé de donner un nom à la constante qui indique le nombre d'éléments du tableau

Exemple :

```
define NBRE 50
int tab[NBRE];
```

**Tableau multi-indicés :**

Un tableau multidimensionnel est un tableau dont les éléments sont eux-mêmes des tableaux.

Le nombre successif d'éléments de chaque dimension est déclaré entre crochets. Les éléments sont rangés consécutivement en mémoire "ligne par ligne".

Exemple :

```
int ecr[2][3];
```

Les éléments du tableau ecr sont rangés en mémoire dans l'ordre :

```
ecr[0][0] ecr[0][1] ... ecr[0][2] ecr[1][0] ... ecr[1][1]
```

Mémoire, Zone ecr
	ecr[0][0], adresse x	...
	ecr[0][1], adresse x + 1	...
	ecr[0][2], adresse x + 2	...
	ecr[1][0], adresse x + 3	...
	ecr[1][1], adresse x + 4	...
	ecr[1][2], adresse x + 5	...
...	...	

Attention:

La déclaration d'un tableau ecr [2] [3] correspond à la création d'un tableau de deux colonnes de trois lignes. Mais la numérotation des cases commence à zéro.

ecr [0] [0]	ecr [1] [0]
ecr [0] [1]	ecr [1] [1]
ecr [0] [2]	ecr [1] [2]

8.3 Utilisation d'une variable tableau

Afin d'utiliser une variable tableau, une fois celle-ci déclarée, il suffit de pointer une case de ce tableau et d'y placer la valeur désirée.

Exemple :

```
ecr[0][1] = 22 ;  
ecr[1][2] = 33 ;
```

Mémoire, Zone ecr
	ecr[0][0], adresse x	...
	ecr[0][1], adresse x + 1	22
	ecr[0][2], adresse x + 2	...
	ecr[1][0], adresse x + 3	...
	ecr[1][1], adresse x + 4	...
	ecr[1][2], adresse x + 5	33
...	...	

Attention :



- Il n'y a pas en C de vérification de débordement de tableau à l'exécution. Ainsi, l'instruction `ecr[0][3]='*'`; écrase le contenu de l'élément `ecr[1][0]`. Ceci constitue une erreur courante, difficile à détecter.

D'autre part, l'instruction `ecr[2][3]='*'`; provoque l'écriture du caractère `*` en dehors de la zone allouée au tableau.

- La notation `ecr[0, 2]` ne désigne pas (comme en Pascal) `ecr[0][2]` mais l'élément `ecr[2]`. Cette erreur ne sera pas détectée par le compilateur (car la virgule est un opérateur).

8.4 Exercices

8.4.1 Gestion de données sous forme d'un tableau

Créer un programme qui demande 4 notes comprises entre 1 et 6 (un chiffre après la virgule) et, qui les placent dans un tableau.

Un test doit être effectué lors de l'entrée des valeurs. Si la valeur n'est pas comprise entre 1 et 6, la valeur est refusée et, le programme demande une nouvelle entrée.

Ensuite, trier le tableau afin de ranger par ordre croissant les valeurs dans le tableau.

Finalement afficher à l'écran les notes par ordre croissant et la moyenne.

Rendre un structogramme avec le code source du programme.



9 Leçon 8 : Classe de stockage, modificateurs et initialisation des variables

9.1 Objectifs de la leçon

Au terme de la leçon, l'élève doit être capable de :

- Décrire le fonctionnement des différentes classes de stockage
- Décrire les différentes méthodes permettant d'initialiser une variable

9.2 Classes de stockage et modificateurs

9.2.1 Classe de stockage d'une variable :

La classe de stockage d'une variable permet de préciser :

- la durée de vie (permanente ou temporaire)
- la visibilité de la variable (partie du programme où cette variable est utilisable).

9.2.2 Variables globales :

- Ce sont des variables déclarées ou définies en dehors de tout bloc (niveau externe).
- Elles ont une durée de vie permanente.
- Elles sont allouées au début de l'exécution du programme et ne sont libérées qu'à la fin de l'exécution du programme.
- Par défaut elles sont visibles dans d'autres modules.
- Elles peuvent être de la classe :
 - extern** : déclare des variables qui sont définies (au niveau externe) dans un autre module
 - static** : limite la visibilité des variables globales au module seul

9.2.3 Variables locales :

Ce sont des variables définies à l'intérieur d'un bloc.

La visibilité de telles variables s'étend uniquement au bloc et aux blocs éventuellement contenus.

Elles peuvent être de classe :

- **auto** : (par défaut) elles sont stockées dans la pile, leur durée de vie est temporaire (limitée à la durée de vie d'exécution du bloc). Elles sont allouées à l'entrée du bloc et libérées à la sortie du bloc.
- **static** : elles sont stockées en zone de donnée statique, leur durée de vie est permanente.
Si une variable de cette classe est initialisée, cette initialisation est réalisée à la compilation du programme.
La classe static permet de préserver la valeur de la variable entre les appels successifs à cette fonction (**variable rémanente**).

**Exemple :**

```
f1() {
    static int i = 0;

    printf("%d\n", i);
    i = i + 1;
}
```

La variable locale *i* (de classe *static*) est initialisée à la compilation avec la valeur 0. Le premier appel à la fonction *f1* affiche 0, le deuxième appel affiche 1 ...

- **register** : seule une variable de type *char*, *int* ou *long* peut utiliser cette classe, qui demande au compilateur d'utiliser si possible un registre du microprocesseur afin d'optimiser l'accès à celle-ci. Sa durée de vie est temporaire. Si l'allocation dans un registre n'est pas possible, la variable est considérée de classe *auto*.

Exemple :

```
/* classe, visibilité et masquage des variables */
int an;                               /* 1 */
static char rep;                       /* 2 */
extern int mode;                       /* 3 */

main()
{
    char str[80];                       /* 4 */
    register int i;                     /* 5 */
    static int jour;                    /* 6 */
    ...
    {
        int i;                          /* 7 */
        ...
    }
    ...
}
```

1. *int an;* : définition d'une variable globale visible dans tous les modules de l'application.
2. *static char rep;* : définition d'une variable globale visible uniquement dans ce module.
3. *extern int mode;* : déclaration d'une variable entière. Celle-ci est définie dans un autre module par `int mode+(au niveau global de cet autre module)`.
4. *char str[80];* : définition d'une variable tableau locale de classe *auto*.
5. *register int i;* : définition d'une variable locale de classe *register*.
6. *static int jour;* : définition d'une variable locale de classe *static*.
7. *int i;* : définition d'une variable locale qui masque la variable locale *i* définie dans le bloc supérieur.

Les variables locales à un bloc masquent les variables de mêmes noms définies à l'extérieur de ce bloc. Les variables masquées conservent leur valeur et redeviennent visibles à la sortie du bloc. L'abondance de masquage peut nuire à la clarté d'un programme.



9.2.4 Modificateurs de type :

Il peut prendre une des valeurs suivantes :

- **const** : informe le compilateur que cette variable ne pourra pas faire l'objet d'une modification de son contenu. Celle-ci n'est accessible qu'en lecture.

Exemple :

```
const int ANNEE = 1992;
```

- **volatile** : informe le compilateur de ne pas faire d'optimisation sur cette variable, parce que cette variable sera modifiée probablement par une cause extérieure (interruption).

9.3 Initialisation des variables

9.3.1 Initialisation explicite

Les variables peuvent être initialisées lors de leur définition. Exemple :

```
int i , j = 5;          /* initialise la variable j a 5 */
char c = 'R';
char tab1[3] = { 'a' , 'b' , 'c' };
char tab2[ ] = { 'a' , 'b' , 'c' };
```

Le compilateur détermine pour tab2 le nombre d'éléments en fonction du nombre d'initialisateurs.

```
int tab3[3][2] = { {1,2},{3,4},{5,6}};
int tab4[3][2] = { 1, 2, 3, 4, 5, 6 }; /* pareil que tab3 */
int tab5[4] = { 1, 2 }; /* tab5[2] = 0 et tab5[3]=0 */
```

S'il y a moins d'initialisateurs que d'éléments déclarés, les éléments non initialisés le sont implicitement à 0.

- En compilation classique (K&R), une variable locale tableau ne pourra être initialisée que si elle est de classe static.
- D'une manière générale, une variable peut être initialisée avec la valeur d'une expression.

Exemples :

```
short a = 0X01 << 3;
int b = 2*a + 3;
int nb = atoi(argv[1]);
```

9.3.2 Initialisation implicite

Elle dépend de la classe :

- classe *extern* ou *static* : ces variables sont permanentes. En l'absence d'initialisation explicite, elles sont initialisées à **0**
- *auto* ou *register* : ces variables ne sont pas initialisées par défaut. Elles contiennent donc une valeur indéfinie.

9.4 Exercices



Finir l'exercice 8.4.1 concernant la Gestion de données sous forme d'un tableau



10 Leçon 9 : Conversions de types

10.1 Objectifs de la leçon

Au terme de la leçon, l'élève doit être capable de :

- Décrire les conversions implicites
- Utiliser la conversion explicite

10.2 Conversions

10.2.1 Conversions implicites :

Le langage C possède ses règles de conversion de type de données à l'intérieur d'une expression.

Le but de ces conversions est d'obtenir une meilleure précision du résultat.

Un certain nombre de conversions de type sont implicitement appliquées :

1. Les opérandes de type *char*, *unsigned char* et *short* sont convertis en *int*
2. Si un opérande est un *long double*, l'autre est converti en *long double*
3. Si un opérande est un *double*, l'autre est converti en *double*
4. Si un opérande est un *float*, l'autre est converti en *float*
5. Des promotions sont effectuées sur les 2 opérandes d'après les règles suivantes :
 1. Si un opérande est un *unsigned long*, l'autre est converti en *unsigned long*
 2. Si un opérande est un *long* et l'autre est un *unsigned*, les 2 opérandes sont convertis en *unsigned long*
 3. Si un des opérandes est un *long*, l'autre est converti en *long*
 4. Si un des opérandes est un *unsigned*, l'autre est converti en *unsigned*
 5. Sinon le résultat est un *int*.

Le résultat sera du type des opérandes.



Il est recommandé de faire un large usage de cet opérateur, même quand une règle de conversion implicite s'applique.

10.3 Exercice

10.3.1 Analyse de texte

Créer un programme qui accepte en entrée une chaîne de 100 caractères maximum et, qui analyse le nombre d'occurrences de certain caractère.

Les caractères testés sont les voyelles (a, e, i, o, u et y), sans tenir compte des minuscule et majuscule.

Ensuite, afficher à l'écran le résultat de votre analyse



11 Leçon 10 : Les pointeurs

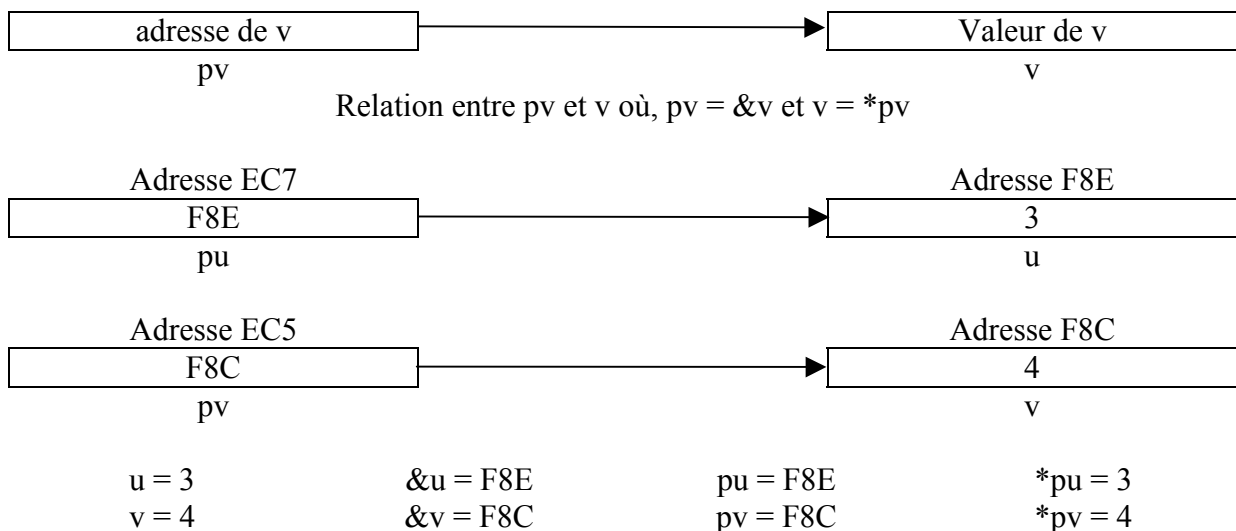
11.1 Objectifs de la leçon

Au terme de la leçon, l'élève doit être capable de :

- Décrire la notions de pointeurs
- Déclarer et utiliser une variable pointeur
- Décrire les opérations arithmétiques appliquées aux pointeurs

11.2 Les pointeurs

Un pointeur est une variable qui représente l'adresse (par opposition à la valeur) d'une variable.



11.2.1 Définition d'une variable pointeur

- La spécificité du langage C vient de son traitement des pointeurs. Leur utilisation est donc très importante.
- Un pointeur est une variable qui contient l'adresse d'une autre variable de n'importe quel type. Sa taille dépend de l'implémentation et est, en général, de la taille d'un `{ int}`.
- La syntaxe générale d'une définition ou déclaration de variable pointeur est de la forme :

classe modificateur type *identificateur ;

La classe, le modificateur et le type sont ceux de la variable pointée.

- Exemple :

```
char *pt1;
```

pt1 est une variable pointeur sur un char. Elle contiendra donc l'adresse d'une zone mémoire capable de contenir elle même un objet de type char.



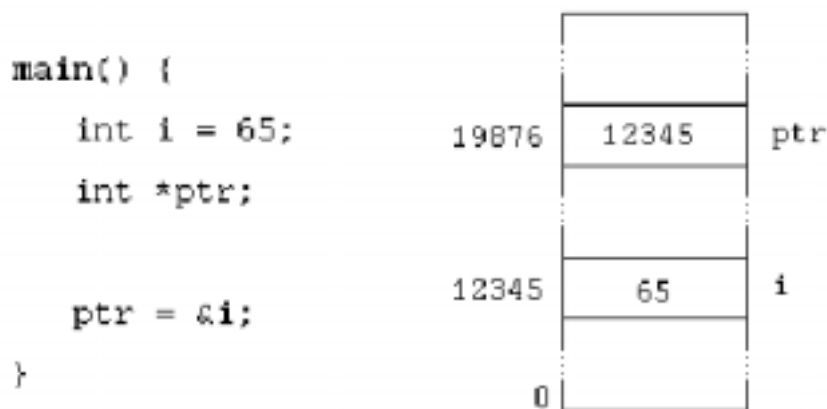
- Un pointeur sur un type void pointe sur une zone sans type.

Exemple :

```
void *pt2;
```

pt2 est une simple adresse mémoire, indépendante du type de l'objet pointé, et qui peut être uniquement affecté à un autre pointeur.

- La définition d'un pointeur n'alloue en mémoire que la place nécessaire pour stocker ce pointeur.
- un pointeur (comme toute autre variable) doit être initialisé avant d'être utilisé.



11.2.2 Opérateur d'adresse

L'opérateur unaire **&** fournit l'adresse en mémoire d'une variable C. Il permet donc d'initialiser la valeur d'un pointeur.

11.2.3 Opérateur d'indirection

- L'opérateur unaire d'indirection ***** (étoile) permet d'obtenir la valeur d'un élément dont l'adresse est contenue dans le pointeur.

Exemple :

```
int i = 65 , *ptr;
```

```
ptr = &i;          /* ptr contient l'adresse de i */  
*ptr = *ptr + 1;  /* equivalent a : i = i + 1; */  
(*ptr)++;        /* equivalent a : i++; */
```

- Il n'est pas possible d'effectuer une indirection sur un pointeur de type void.

Exemple :

```
void *ptr1;  
int *ptr2;
```

```
/* initialisation de ptr1 et ptr2 : */  
PTR1 = .....;
```



```
ptr2 = .....;

*ptr1 = 10; /* INCORRECT */
ptr2 = (int *) ptr1; /* CORRECT */
```

Je rappelle, qu'un pointeur (comme toute autre variable) doit être initialisé avant d'être utilisé. Donc avant d'écrire **ptr1* il faut donner une valeur au pointeur.

11.3 Arithmétique des pointeurs

11.3.1 Incrémentation/décrémentation de pointeur :

L'incrémentation (ou la décrémentation) de pointeur exprime le fait que l'on veuille atteindre l'élément pointé suivant (ou précédent).

Le résultat est correct que si l'objet pointé est situé dans un même tableau.

Ainsi avec la définition suivante :

```
int *ptr;
```

les instructions **ptr++**; ou **ptr = ptr + 1**; sont équivalentes.

Elles incrémentent la valeur du pointeur de la longueur du type de l'objet pointé, permettant ainsi de désigner le début de l'élément suivant.

Ces opérations sont impossibles avec des pointeurs sur un type void.

11.3.2 Addition/soustraction de 2 pointeurs :

- l'addition de 2 pointeurs n'a pas de sens et est donc sans signification.
- la soustraction de 2 pointeurs (de même type) situés dans un même tableau, retourne le nombre d'éléments qui les séparent.

Exemple :

```
int tab[20] , *pt1 = &tab[2] , *pt2 = &tab[6];
printf("%d\n", pt2 - pt1); /* affiche : 4 */
```

11.3.3 Comparaison de pointeurs :

Un pointeur ne pourra être comparé qu'à un pointeur sur le même type (ou à la constante NULL). Le résultat de la comparaison sera significatif et portable que si les pointeurs pointent sur le même objet.

11.3.4 Affectation de pointeurs :

On ne doit affecter à un pointeur qu'une valeur de pointeur sur un objet de même type. Dans le cas contraire, de telles affectations risquent de ne pas être portable (contrainte d'alignement,

représentation des données en mémoire ...). On utilisera une conversion explicite (casting) dans ce cas.

**Exemple :**

La fonction d'allocation dynamique malloc retourne un pointeur sur un void : si l'on veut faire une allocation dynamique d'un entier, on doit écrire (pour être portable) :

```
int *t; /* t est un pointeur sur un entier */
t = (int *) malloc( sizeof(int));
```

t contient l'adresse de début d'une zone de mémoire pouvant contenir un entier.

11.4 Pointeurs et tableaux

- Le nom seul d'un tableau est un **pointeur constant** sur le premier élément de celui-ci.

Avec les définitions ci dessous :

```
char *ptab;      /* ptab est un pointeur sur un char */
char tab[10];   /* tableau de 10 char */
```

on peut écrire :

- `ptab = &tab[0]`; qui est équivalent à : `ptab = tab`;
- `tab[0] = 'a'`; qui est équivalent à : `*ptab = 'a'`;
- `tab[3] = 'd'`; qui est équivalent à : `*(ptab+3) = 'd'`;

ainsi on peut dire que :

```
identif[expression]
```

est équivalent à :

```
*(identif + expression )
```

- L'affectation globale entre tableaux n'est pas possible. Ainsi après la définition suivante :

```
int tab1[80], tab2[80];
```

l'instruction `tab1 = tab2`; n'est pas possible car le nom seul d'un tableau est un pointeur constant sur le premier élément de celui-ci.

Cette écriture n'a pas plus de sens que l'expression : $2 = 3$

- En C, une chaîne de caractères est considérée comme un tableau de caractères, dont le dernier élément est le caractère `'\0'` (caractère de code 0).
- Pour travailler sur un tableau on a donc le choix entre travailler avec un tableau en utilisant un indice ou travailler avec un pointeur.

Exemples :

```
/* copie de chaine, version tableau */
void copie(char srce[] , char dest[])
{
```



```
register int i; /* un indice est necessaire */
for( i=0 ; (dest[i] = srce[i]) != 0 ; i++)
    ;
}

/* copie de chaine, version pointeur */
void copie(char *srce , char *dest)
{
    while ( (*dest++ = *srce++) != 0 )
        ;
}
```

11.5 Les tableaux et les chaînes

L'initialisation d'un tableau de caractères peut se faire de la manière suivante :

```
char msg[] = "Hello";
```

Cette écriture est l'abréviation de :

```
char msg[] = {'H','e','l','l','o','\0'};
```

La dimension du tableau est ici facultative, car elle peut être calculée par le compilateur.

msg est un tableau initialisé, on ne peut donc modifier que son contenu.

En dehors de la déclaration, il n'est pas possible d'affecter une chaîne de caractères à un tableau. Il faut passer par une fonction de copie de chaînes de caractères (*strcpy* par exemple).

Par contre dans la définition suivante :

```
char *adieu = "Au revoir";
```

adieu est un pointeur qui contient l'adresse du premier caractère de la **constante** chaîne de caractères "Au revoir".

On ne peut donc pas modifier cette chaîne.

D'autre part :

```
char ligne[81];
```

```
ligne = "Bonjour"; /* INTERDIT !!! */
```

ligne est le nom d'un tableau, c'est un pointeur constant et ce n'est donc pas une "lvalue".

"Bonjour" est une **constante** chaîne de caractères et est considérée comme une constante de type tableau de caractères.

Il est aussi abusif d'écrire *ligne = "Bonjour"*; que d'écrire $2 = 3$; ou $2 = a$; !!!

Il faut employer *strcpy(ligne, "Bonjour");* (ou la fonction *copie* de l'exemple précédent).



11.6 Exercices

11.6.1 Les pointeurs

Un programme en C contient les instructions suivantes :

```
char u, v = 10 ;
char *pu, *pv = &v;
...
*pv = v+1 ;
u = *pv + 1 ;
pu = &u;
```

En supposant qu'un caractère occupe un octet, que la valeur de u soit rangée à l'adresse F8C (hexadécimal), et que v soit situé à l'adresse F8D :

- Quelle est la valeur représentée par &v ?
- Quelle est la valeur affectée à pv ?
- Quelle est la valeur représentée par *pv ?
- Quelle est la valeur affectée à u ?
- Quelle est la valeur représentée par &u ?
- Quelle est la valeur affectée à pu ?
- Quelle est la valeur représentée par *pu ?

Finir l'exercice 10.4.1 concernant l'analyse de texte
--



12 Leçon 11 : Les fonctions

12.1 Objectifs de la leçon

Au terme de la leçon, l'élève doit être capable de :

- Décrire la notion de fonctions
- Créer un fonction contenant des entrées et, des sorties
- Utiliser une fonctions à partir du programme principal

12.2 Les fonctions

12.2.1 Généralités

Les fonctions sont les briques de base d'un programme C. Elles nous permettent de découper des programmes en parties plus petites donc plus faciles à relire et à mettre au point.

Une fonction est un sous-programme qui effectue un travail bien précis, à laquelle on passe (généralement) des données et qui retourne (le plus souvent) une valeur.

Cette capacité nous permet de créer des routines indépendantes qui pourront être appelées depuis d'autres fonctions.

Quand une fonction est appelée, l'exécution du programme est transférée à la première instruction de cette fonction. L'exécution de la fonction se termine après l'instruction *return* ou après la dernière instruction du bloc de la fonction.

Quand la fonction appelée a terminé, l'exécution dans la fonction appelante, reprend à l'endroit de l'expression où l'appel avait été fait. La valeur retournée par la fonction peut être alors utilisée comme opérande dans cette expression.

L'échange de données entre fonctions est réalisé par l'intermédiaire de variables globales et/ou par une liste d'arguments. L'échange par variables globales n'est à effectuer que dans de très rares occasions.

12.2.2 Passage des paramètres

Quand on invoque une fonction, on doit généralement lui passer des valeurs.

Il y a deux points de vue pour le passage des données à la fonction, celui de la fonction appelante et celui de la fonction appelée.

- Du côté appelant, les valeurs passées sont appelées **paramètres réels** ou **paramètres effectifs** ou **arguments**.

Par exemple, dans l'appel suivant, deux arguments, une variable (*s1*) et une constante chaîne de caractères, sont passés :

```
strcpy( s1 , "Le langage C" );
```



- Du côté appelé, les variables spécifiées sont appelées **paramètres formels** ou **paramètres**.
- Lors de l'appel de la fonction, la liste des paramètres réels est mise en correspondance avec la liste des paramètres formels.
- Le mode de transmission entre ces deux contextes est effectué **par valeur** :

la fonction travaille donc sur une copie des paramètres réels.

Et la modification d'une copie ne modifie pas l'original...

Attention :

- En compilation classique, le nombre et le type des paramètres effectifs ne sont pas contrôlés.
- L'ordre d'évaluation des paramètres n'est pas défini dans le langage. Ainsi dans l'instruction suivante :

```
toto ( ++i , tab[i] );
```

est-ce *i* ou *i* incrémenté de 1 qui est l'index de *tab* ?

12.2.3 L'appel d'une fonction

- L'appel d'une fonction est réalisé en signifiant le nom de la fonction suivi de la liste des paramètres réels.
- L'appel de la fonction suppose que celle-ci ait été définie ou déclarée auparavant. Dans le cas contraire C suppose que la valeur retournée est du type *int* et aucune erreur n'est signalée...

Exemple :

```
b = theta ( 'a' , 180 , v );  
pi2 = 2 * pi();  
cls();
```

12.2.4 Définition ANSI d'une fonction

- **Syntaxe :**

```
classe type identificateur  
    (liste_typée_paramètres_formels) bloc_des_instructions
```

- **classe** : classe d'une fonction.
Par défaut, la fonction est visible dans tous les modules de l'application.
 - *extern* : permet de déclarer une fonction qui est définie dans un autre module de l'application.



- *static* : la fonction n'est visible que dans le module dans lequel elle est définie.
- *type* : (facultatif) type de la valeur retournée par la fonction. Par défaut c'est le type *int* qui est retourné.
Le type *void* précise que la fonction ne retourne pas de valeur.
- *liste_typée_paramètres_formels* : c'est la liste de déclaration des paramètres formels de la fonction.

- **Exemple :**

```
float theta( char cas , int angle , float vitesse )
{
    instruction1;
    ...
}
```

12.2.5 Déclaration ANSI d'une fonction

- **Syntaxe :**

```
classe type identificateur(
    liste_typée_paramètres_formels ) ;
```

Pour déclarer une fonction, il suffit de reprendre son entête et de mettre un ; (point virgule) pour finir l'instruction.

- **Exemple de déclaration de fonctions :**

```
float theta( char cas , int angle , float vitesse ) ;
double pi(void);
void cls( void ) ;
```

12.2.6 Les prototypes

On nomme **prototype**, une déclaration complète de fonction qui comprend donc :

- Le type de la valeur retournée par la fonction
- la liste de déclaration des paramètres formels, qui peuvent avoir un des formats suivants :
 - *type* : seul le type est utilisé par le compilateur pour mettre en place les conversions nécessaires
Exemple : float theta(char , int , float);
 - *type nom_parametre* : les identificateurs *nom_parametre* sont purement fictifs et ne servent qu'à donner au prototype une forme comparable à l'en-tête de la fonction.
Exemple : int somme(int a , int b);

Le prototype est utilisé par le compilateur pour mettre en place les conversions nécessaires et vérifier que les paramètres réels correspondent bien en nombre et en type aux paramètres formels.

En l'absence de prototype le compilateur mettra en oeuvre les conversions par défaut.



Il est possible de passer un *char* ou un *float*, par exemple, en paramètre sans qu'ils soient convertis respectivement en *int* et *double* dans le cas d'une déclaration classique (conversions par défaut).

12.2.7 Passage par "référence"

Ce mode de passage, contrairement à d'autres langages, n'existe pas en C. Le seul mode de passage des paramètres est le mode de passage "par valeur". De plus, une fonction ne peut retourner qu'une seule valeur.

Si l'on désire écrire une fonction qui puisse :

- nous fournir plusieurs résultats,
- nous fournir un résultat dans un de ses paramètres réels,
- modifier un paramètre d'entrée,

il faut :

- **définir une structure pour les valeurs de retour de la fonction**

Exemple de déclaration d'une fonction qui doit nous fournir les valeurs *min* et *max* d'un tableau d'entiers :

```
int *min_max(int tab[], int dim);
/* retourne un tableau de 2 entiers contenant le min et
le max */
/* entraine un problème d'allocation des valeurs
retournées ... */
```

- **fragmenter les fonctions**, de sorte qu'elles n'aient qu'une seule valeur de retour chacune

```
int min(int tab[], int dim);
int max(int tab[], int dim);
```

ou

```
int min_max(int tab[], int dim, int flag);
/* si flag = 0 min_max() retourne la valeur min */
/* si flag = 1 min_max() retourne la valeur max */
```

- **passer par des pointeurs :**

on transmet un pointeur sur l'objet en question : le paramètre formel fait alors référence à la même position mémoire que l'argument.

Si la fonction change la valeur contenue à cette adresse, elle change aussi la valeur de l'argument passé.

```
void min_max(int tab[], int dim, int *min, int *max);
```

Les deux premières solutions ne sont pas élégantes et c'est la troisième solution qui est utilisée par les programmeurs C.

**Exemple : échange du contenu de 2 variables**

```
#include <stdio.h>

void echange_rien(int a , int b)
{
    int temp;
    temp = a; a = b; b = temp;
}

void echange_bien(int *pa , int *pb) /* echange de 2 entiers */
{
    int temp;
    temp = *pa; *pa = *pb; *pb = temp;
}

main()
{
    int x = 10 , y = 20;

    echange_rien( x , y );
    printf("x = %d y = %d\n", x, y); /* affichage: x = 10 y = 20 */

    echange_bien( &x , &y );
    printf("x = %d y = %d\n", x, y); /* affichage: x = 20 y = 10 */
}
```

12.2.8 L'instruction return

Cette instruction :

1. permet de renvoyer la valeur précisée en argument à l'instruction appelante.
2. provoque une sortie immédiate du bloc principal de la fonction.

Exemple :

```
return 12;
return ; /* retourne une valeur aléatoire */
return ( a > b ) ? a : b ;
```

12.2.9 Arguments de main()

Les arguments, de la ligne de commande qui lance l'exécution d'un programme, sont fournis au programme sous la forme d'un tableau de pointeurs.

Pour pouvoir les utiliser, il faut déclarer la fonction *main* de la façon suivante :

```
main(int argc , char *argv[ ])
```

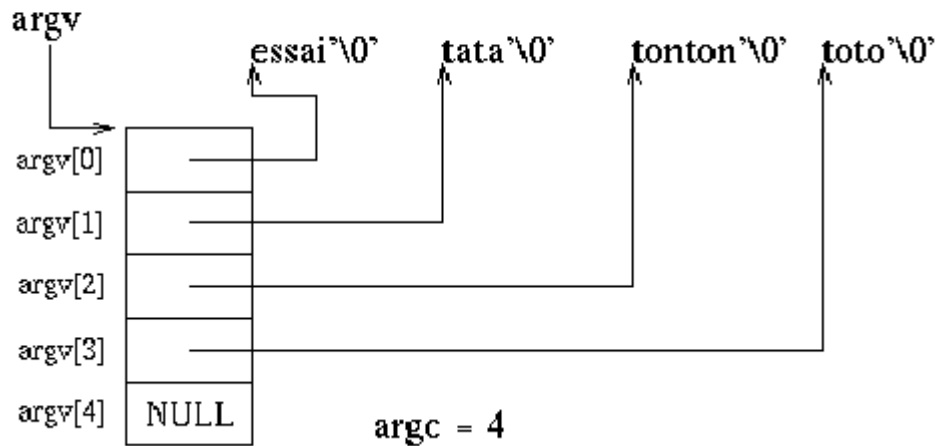
- *argc* contient le nombre d'arguments de la ligne de commande.
- *argv* est un tableau de pointeurs sur les arguments de la ligne de commande shell.

**Remarques :**

- *argv[0]* pointe sur le nom de la commande, il est donc toujours le nom du programme lui même (et donc *argc* est toujours au moins égal à 1).
- *argc* et *argv* sont des noms conventionnels.
- La norme précise que *argv[argc]* doit être un pointeur *NULL*.

ligne de commande :

apollo25:[~]> essai tata tonton toto

**Exemples :**

/* affiche les arguments de la ligne de commande, version tableau */

```
int main(int argc, char *argv[])
{
    register int i;
    for ( i = 0 ; i < argc ; i++ )
        printf("%s\n", argv[i]);
}
```

/*affiche les arguments de la ligne de commande, version pointeur */

```
int main(int argc, char **argv)
{
    /* **argv équivaleant: *argv[] équivaleant: argv[][] */
    while ( argc-- )
        printf("%s\n", *argv++);
}
```



12.3 Exercice

12.3.1 Programmation efficace à l'aide des fonctions

Créer un petit programme qui calcul la puissance 3 d'un nombre.

Le calcul de la mise au cube doit être contenue dans une fonction.

Le calcul s'effectue sur 2 variable initialisée dans le programme.

Finalement, sortir les résultat à l'écran sous la forme :

```
3 à la puissance trois égal 27
```



13 Leçon 12 : Opérateurs de manipulation de bits et, structures

13.1 Objectifs de la leçon

Au terme de la leçon, l'élève doit être capable de :

- Décrire les similitudes entre la manipulation des bits en C et en électronique numérique
- Utiliser les opérations sur les bits et calculer le résultat attendu
- Décrire la notion de structure

13.2 Opérateurs de manipulation de bits

Les opérateurs qui suivent opèrent bit à bit et s'appliquent à des opérandes de type entier (ou caractère) et de préférence *unsigned* (sinon ils risquent de modifier le bit de signe).

Ils procurent des possibilités de manipulation de bas-niveau de valeurs, traditionnellement réservées à la programmation en langage assembleur.

opérateur	opération
~	NEGATION BIT A BIT (UNAIRE)
<<	décalage à gauche
>>	décalage à droite
&	ET bit à bit
	OU (inclusif) bit à bit
^	OU exclusif bit à bit

- ~ : négation bit à bit (opérateur unaire). Il inverse un à un tous les bits de son unique opérande.

Exemple : 0x5F est une expression qui vaut : 160 (0xA0)

```
0x5F      0 1 0 1   1 1 1 1
~0x5F     1 0 1 0   0 0 0 0   -> 0xA0   -> 160
```

- << : décalage à gauche. $a \ll 3$ décale de 3 rangs à gauche la valeur contenue dans a . Les bits sortants à gauche sont perdus et des 0 sont introduits à droite.

Cet opérateur permet d'effectuer des multiplications d'entiers par des puissances de 2. Si la variable est signée, le bit de signe est conservé.

- >> : décalage à droite. $a \gg 3$ décale de 3 rangs à droite la valeur contenue dans a . Les bits sortants à droite sont perdus et des 0 sont introduits à gauche.

Cet opérateur permet d'effectuer des divisions d'entiers par des puissances de 2. Si la variable est signée, le bit de signe est conservé et propagé.

**Exemple :**

```
int i = -100;
printf("%d\n", i >> 2 );           /* affichage de -25 */
printf("%d\n", i << 2 );           /* affichage de -400 */
```

- **&** : ET bit à bit entre les valeurs de 2 expressions.
a & b chaque bit du résultat vaut 1 si les bits de même rang de a et b valent 1, 0 sinon.
- **|** : OU (inclusif) bit à bit entre les valeurs de 2 expressions.
a | b chaque bit du résultat vaut 1 si les bits de même rang de a ou b valent 1, 0 sinon.
- **^** : OU exclusif bit à bit entre les valeurs de 2 expressions.
a ^ b chaque bit du résultat vaut 1 si les bits de même rang de a et b sont différents, 0 sinon.

Table de vérité des opérateurs bit à bit :

op1	op2	op1 & op2	op1 op2	op1 ^ op2
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Exemple : 0xB6 & 0x53 est une expression qui vaut 18 (0x12) parce que :

	1 0 1 1	0 1 1 0		0xB6
&			&	
	0 1 0 1	0 0 1 1		0x53
	-----			-----
	0 0 0 1	0 0 1 0	->	0x12 -> 18

13.3 Les structures

Les tableaux permettent de désigner sous un même nom un ensemble de valeurs de même type.

Les structures permettent de regrouper des objets de types hétérogènes.

Une variable de type structure est une variable composée de champs, eux-mêmes déclarés d'un certain type.

- Déclaration de type structure
- Utilisation des champs d'une structure
- Structures récursives (ou autoréférentielles)
- const et les structures



13.3.1 Déclaration de type structure :

Une déclaration de type structure ne définit aucune variable, elle permet de définir un modèle de structure.

Syntaxe :

```
struct identificateur_de_type
{
    type identificateur_de_champ;
    type identificateur_de_champ;
    ...
};
```

Exemple :

```
struct s_fiche
{
    char nom[30];
    char sexe;
    int numero;
};
```

On peut donc définir des variables ayant ce type :

```
/* definition de etat_civil de type s_fiche */
struct s_fiche etat_civil;

/* definition et initialisation */
struct s_fiche fiche = {"Ritchie", 'M', 65};

/* tableau de 100 structures de type s_fiche */
struct s_fiche fichier[100];
```

La déclaration d'une variable structure peut suivre immédiatement la définition du type :

```
struct identificateur_de_type
{
    type identificateur_de_champ;
    type identificateur_de_champ;
    ...
} identificateur_de_variable ... ;
```

Exemple :

```
struct t_fiche
{
    char nom[30];
    char sexe;
    int numero;
} etat_civil , fiche;
```

Les champs d'une structure peuvent être de n'importe quel type (sauf elle même ou une fonction).



Attention : pour connaître la taille d'une variable de type structure, il faut impérativement utiliser l'opérateur sizeof.

13.3.2 Utilisation des champs d'une structure :

Pour faire référence à un champ particulier de la structure, on accole le nom de la structure concernée avec le nom du champ visé, et on insère le caractère . (point) entre les 2 termes.

Exemple :

```
strcpy(etat_civil.nom , "Thompson" ) ;  
etat_civil.sexe = 'M' ;
```

On peut aussi faire référence au champ à l'aide d'un pointeur :

```
/* definition d'une variable pointeur sur une structure de type  
s_fiche */  
struct s_fiche *membre;  
...  
/* accès au champ nom */  
membre->nom /* equivalent : (*membre).nom */
```

l'opérateur de pointeur de structure -> (symbole - (moins) suivi du symbole > (supérieur)) pointe vers un membre particulier de la structure.

Il est possible d'affecter à une structure le contenu d'une autre structure définie avec le même modèle.

```
struct s_fiche fiche1 , fiche2;  
fiche1 = fiche2;
```

Exemple :

```
#include <stdio.h>  
  
struct s_date  
{  
    int jour;  
    int mois;  
    int an;  
};  
  
struct s_fiche  
{  
    char nom[20];  
    int numero;  
    struct s_date naissance;  
};  
  
void afficher( struct s_fiche f );  
struct s_date SaisirDate( void );  
void SaisirFiche( struct s_fiche *f );
```



```
/*----- programme principal -----*/
main()
{
    struct s_fiche fiche;

    SaisirFiche(&fiche);
    afficher(fiche);
    return 0;
}

/*-----*/
void afficher( struct s_fiche f )
{
    printf("Nom : %s\n", f.nom );
    printf("Numero : %d\n", f.numero);
    printf("Date de naissance : %d/%d/%d\n",
        f.naissance.jour, f.naissance.mois,
        f.naissance.an );
}

/*-----*/
struct s_date SaisirDate( void )
{
    struct s_date date;

    printf("Jour ? ");
    scanf("%d", &date.jour); fflush(stdin);
    printf("Mois ? ");
    scanf("%d", &date.mois); fflush(stdin);
    printf("Annee ? ");
    scanf("%d", &date.an); fflush(stdin);
    return date;
}

/*-----*/
void SaisirFiche( struct s_fiche *f )
{
    printf("Nom ? ");
    gets( f->nom );
    printf("Numero ? ");
    scanf("%d", &f->numero); fflush(stdin);
    f->naissance = SaisirDate();
}
}
```



13.4 Exercice

13.4.1 Manipulations de bits

Effectuer les opérations suivantes, pour a valant 0xa2c3 :

- a) $\sim a$ résultat =
- b) $a \& 0x3f06$ résultat =
- c) $a \wedge 0x3f06$ résultat =
- d) $a | 0x3f06$ résultat =
- e) $a \ll 5$ résultat =



14 Projets

14.1 Graphique d'une fonction mathématique

(temps : 8 leçons)

Etudier la fonction 'Line' de la librairie 'Graphics.h'.

Ensuite, créer un programme qui va tracer à l'écran la courbe de la fonction :

$$F(x) = a + b * x + c * x^2 + d * x^3$$

Le programme demande les valeurs de a, b, c et d, trace le graphique pour des valeurs de x allant de 0 à 50.

Le calcul de l'échelle doit se faire automatiquement et, figurer sur les axes du graphique.

14.2 Calcul du prix du transport

(temps : 4 leçons)

Construire les structogrammes permettant l'affichage du prix d'un billet de transport.

Le déroulement des opérations est le suivant :

1. Affichage du titre "CALCUL DU PRIX DU TRANSPORT".
2. Saisie de la distance à parcourir en km.
3. Le calcul proprement dit du prix du transport.
4. L'affichage du tarif à payer.
5. Une invite à recommencer ou quitter.

Le prix du billet dépend de la distance donnée au programme :

- les 10 premiers kilomètres sont au tarif de 50 cts le kilomètre.
- du 11e au 50e kilomètre, le tarif est de 30 cts le kilomètre.
- du 51e au 100e kilomètre, le tarif est de 20 cts le kilomètre.
- dès le 101e kilomètre, le tarif est de 10 cts le kilomètre.



Exemple d'écran lors de l'utilisation du logiciel :

CALCUL DU TARIF D'UN BILLET DE TRANSPORT

Entrer la distance : 124 Km

Tarif du transport : 29.40 frs

Voulez-vous un autre calcul O/N ?

Le travail rendu devra contenir au minimum :

- A) Un structogramme principal du déroulement général du logiciel
- B) Un sous-structogramme qui indique comment le programme choisi de calculer le tarif en fonction de la distance indiquée.
- C) Les sous-structogrammes qui contiennent les calculs du prix proprement dit selon les cas de figure donnés par B).

14.3 Cercle

(temps : 4 leçons)

Construire un programme permettant de :

- dessiner un cercle de rayon donné au centre de l'écran
- déterminer la surface ainsi que le périmètre de ce cercle

Utiliser une fonction du C pour dessiner un cercle.

Le programme doit demander le rayon du cercle en millimètre et doit contrôler que ce cercle entre sur l'écran pour l'échelle 1 pixel = 1 mm.



14.4 Carnet d'adresses

(temps : 4 leçons)

Réaliser un programme en C permettant de gérer votre carnet d'adresses en utilisant un tableau. On suppose qu'une adresse est constituée des éléments suivants :

- Nom : 15 caractères
- Rue : 20 caractères
- Numéro : de 1 à 199
- Localité : 25 caractères
- Téléphone : 10 caractères

Les fonctions à implanter sont les suivantes :

- Ajout d'adresses
- Rechercher par nom.

14.5 Gestion des I/O d'un PC

(temps : 8 leçons)

Utilisations du module I/O pour réaliser un jeu de lumières programmable.

Etudier la fonction 'Time' de la librairie 'Time.h'.

Le déplacement des LED's est contrôlé par les switches de la manière suivante.

SW1	Déplacement de gauche à droite à la vitesse programmée	Priorité 2
SW2	Déplacement de droite à gauche à la vitesse programmée	Priorité 3
SW3	Clignotement de toutes les LED's ensembles à la vitesse programmée	Priorité 4
SW4	Toutes les LED's éteintes	Priorité 1
SW5	Programmation du temps : 1 seconde	
SW6	Programmation du temps : 2 secondes	
SW7	Programmation du temps : 4 secondes	
SW8	Programmation du temps : 8 secondes	

Le contrôle du port parallèle se fait à l'aide des fonctions 'Entree_pp' et 'Sortie_PP' fournie par le maître.

L'état des LED's et des switches doit apparaître sur l'écran, sous la forme :

Contrôle des I/O du PC

Déplacement vers la gauche à la vitesse de un cran pour 10 secondes

O O X O O O O O



14.6 Gestion des feux d'un carrefour

(temps : 8 leçons)

Utilisations du module I/O pour réaliser la gestion d'un carrefour.

Etudier la fonction 'Time' de la librairie 'Time.h'.

Les LED's vertes et rouges représentent les feux de signalisations et, les switchs simulent les capteurs magnétiques signalent la présence d'une voiture.

Etudier les différents cas possibles et, des décisions à prendre et, faites en un tableau.

Lorsque aucune voiture attend, la route principale doit avoir ces feux verts. Ensuite, créer un schéma qui va permettre de gérer la circulation de la manière la plus fluide possible.

Dans la cas d'un seul usager dans le carrefour, son feu doit passer au vert au plus tard 2 secondes après son arrivée.

Dans le cas ou le carrefour est utilisé par un grand nombre d'utilisateurs, le feu doit rester vert 10 secondes.

L'intervalle entre deux feux vert et de 5 secondes, de façon à libérer le carrefour.

(La voiture met 15 secondes pour aller du carrefour n° 1 au carrefour n° 2)

Le contrôle du port parallèle se fait à l'aide des fonctions 'Entree_PP' et 'Sortie_PP' fournie par le maître.

L'état des feux et des capteurs de présence doit apparaître à l'écran, sous la forme

Gestion des feux d'un carrefour :

Carrefour n° 1

Feu n°1 :	Voiture en attente	Rouge
Feu n°2 :	Aucune voiture	Rouge
Feu n°3 :	Voiture en attente	Vert
Feu n°4 :	Voiture en attente	Rouge
Feu n°5 :	Aucune voiture	Rouge
Feu n°6 :	Voiture en attente	Rouge
Feu n°7 :	Voiture en attente	Vert
Feu n°8 :	Aucune voiture	Rouge