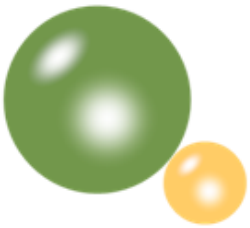
Programmation Structurée 1

Ecole d'Ingénierie

Première année



Les bases de la programmation en C

Table des matières

1. Définitions

Bit (Binary digiT) ou élément binaire, la plus petite partie manipulable par un ordinateur. Comme son nom l'indique un élément binaire peut prendre deux valeurs : 0 ou 1.

Octet ou byte, le groupe de bits qui permet de supporter la représentation d'un caractère. Ce groupe est le plus souvent (comme son nom l'indique) constitué de huit bits. En langage C, l'octet sert dans les calculs d'occupation d'espace mémoire et en particulier les variables plus compliquées comme le mot machine ont une taille donnée en nombre d'octets.

Pile ou pile d'exécution, c'est une partie de l'espace mémoire d'une application qui permet au programme de tenir à jour des séries de variables actives. Cet espace est géré comme une pile d'assiettes, c'est-à-dire que l'on peut ajouter de l'espace (faire grandir la pile) ou diminuer cet espace seulement par le haut. Ainsi, les derniers éléments qui ont été ajoutés sur une pile sont les plus facilement accessible, une pile représente le modèle **LIFO** (Last In First Out) les derniers éléments qui y sont ajoutés sont les premiers à pouvoir être retirés. Les opérations d'agrandissement ou de réduction de la pile sont faites de manière automatique lors des appels de fonctions et respectivement des retours de fonctions. La pile est gérée à partir d'un espace mémoire de taille fixe qui est attribué de manière automatique par le système d'exploitation. Le processeur tient à jour sa relation avec la pile à travers un registre interne qui décrit le sommet de pile et un autre registre qui maintient un lien sur le contexte (arguments de l'appel et variables locales).

Le préprocesseur C ou cpp assure une phase préliminaire de la traduction (compilation) des programmes informatiques écrits dans les langages de programmation C et C++. Comme préprocesseur, il permet principalement l'inclusion d'un segment de code source disponible dans un autre fichier (fichiers d'en-tête ou header), la substitution de chaînes de caractères (macro définition), ainsi que la compilation conditionnelle.
Dans de nombreux cas, il s'agit d'un programme distinct du compilateur lui-même et appelé par celui-ci au début de la traduction. Le langage utilisé pour les directives du préprocesseur est indépendant de la syntaxe du langage C, de sorte que le préprocesseur C peut être utilisé isolément pour traiter d'autres types de fichiers sources.

Le langage Assembleur ou langage d'assemblage, dit assembleur tout court, est le langage de programmation le plus proche - tout en restant lisible par un être humain - du langage machine utilisé par le (micro)processeur de la machine. Le langage machine exprime les instructions des programmes par des combinaisons de bits, comme :

```
10110000 01100001
```

En langage d'assemblage, ces instructions sont représentées par des symboles mnémoniques bien plus lisibles, comme :

```
mov $0x61, %al
```

(qui signifie de mettre la valeur hexadécimale 61 (97 en décimal) dans le registre 'AL'.)

Contrairement à un langage de haut-niveau, il y a une correspondance 1-1 entre le code assembleur et le langage machine, ainsi les ordinateurs peuvent traduire le code dans les 2 sens sans perdre d'information. La transformation du code assembleur dans le langage machine est accomplie par un programme nommé Assembleur, dans l'autre sens par un programme Désassembleur. L'opération s'appelle respectivement, assemblage et désassemblage.

Chaque architecture d'ordinateurs a son propre langage machine, et donc son propre langage d'assemblage (l'exemple ci-dessus est pour l'architecture x86 des PC). Ces différents langages diffèrent par le nombre et le type d'opérations qu'ils ont à supporter. Ils peuvent avoir des tailles et des nombres de registres différents, et différentes représentations de type de données en mémoire. Tandis que tous les ordinateurs sont capables de faire les mêmes choses, ils le font de manière différente.

Par ailleurs, il peut exister plusieurs syntaxes pour le même jeu d'instruction, selon les conventions choisies par le fournisseur du programme d'assemblage. L'exemple ci-dessus était donné en syntaxe AT&T, en syntaxe Intel il s'écrit :

```
MOV AL,61h
```

2. Historique

Le C a été conçu en 1972 par Dennis Richie et Ken Thompson, chercheurs aux Bell Labs, afin de développer un système d'exploitation UNIX sur un DEC PDP-11. En 1978, Brian
Kernighan et Dennis Richie publient la définition classique du C dans le livre *The C Programming language* [6].
Le C devenant de plus en plus populaire dans les années 80, plusieurs groupes mirent sur le marché des compilateurs comportant des extensions particulières. En 1983, l'ANSI (American National Standards Institute) décida de normaliser le langage ; ce travail s'acheva en 1989 par la définition de la norme ANSI C. Celle-ci fut reprise telle quelle par l'ISO (International Standards Organization) en 1990. C'est ce standard, ANSI C, qui est décrit dans le présent cours.

3. Compilation

Le C est un langage compilé (par opposition aux langages interprétés). Cela signifie qu'un programme C est décrit par un fichier texte, appelé fichier source. Ce fichier n'étant évidemment pas exécutable par le microprocesseur, il faut le traduire en langage machine. Cette opération est effectuée par un programme appelé compilateur. La compilation se décompose en fait en 4 phases successives :

1. Le traitement par le préprocesseur : le fichier source est analysé par le préprocesseur qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers source . . .).
2. La compilation : la compilation proprement dite traduit le fichier généré par le préprocesseur en assembleur, c'est-à-dire en une suite d'instructions du microprocesseur qui utilisent des mnémoniques rendant la lecture possible.
3. L'assemblage : cette opération transforme le code assembleur en un fichier binaire, c'est-à-dire en instructions directement compréhensibles par le processeur. Généralement, la

compilation et l'assemblage se font dans la foulée, sauf si l'on spécifie explicitement que l'on veut le code assembleur. Le fichier produit par l'assemblage est appelé fichier objet.

4. L'´edition de liens : un programme est souvent séparé en plusieurs fichiers source, pour des raisons de clarté mais aussi parce qu'il fait généralement appel à des librairies de fonctions standard déjà écrites. Une fois chaque code source assemblé, il faut donc lier entre eux les différents fichiers objets. L'´edition de liens produit alors un fichier dit exécutable.

Les différents types de fichiers utilisés lors de la compilation sont distingués par leur suffixe.
Les fichiers source sont suffixés par .c, les fichiers prétraités par le préprocesseur par .i, les fichiers assembleur par .s, et les fichiers objet par .o.

4. Les composants élémentaires du C

Un programme en langage C est constitué des six groupes de composants élémentaires suivants :

- Les identificateurs,
- Les mots-clefs,
- Les constantes,
- Les chaînes de caractères,
- Les opérateurs,
- Les signes de ponctuation.

On peut ajouter à ces six groupes les commentaires, qui sont enlevés par le préprocesseur.

4.1 Les identificateurs

Le rôle d'un identificateur est de donner un nom à une entité du programme. Plus précisément, un identificateur peut désigner :

- un nom de variable ou de fonction,
- un type défini par typedef, struct, union ou enum,
- une étiquette.

Un identificateur est une suite de caractères parmi :

- les lettres (minuscules ou majuscules, mais non accentuées),
- les chiffres,
- le "blanc souligné" (_).

Le premier caractère d'un identificateur ne peut pas être un chiffre. Par exemple, var1, tab 23 ou deb sont des identificateurs valides ; par contre, 1i et i:j ne le sont pas. Il est cependant déconseillé d'utiliser comme premier caractère d'un identificateur car il est souvent employé pour définir les variables globales de l'environnement C.
Les majuscules et minuscules sont différenciées.
Le compilateur peut tronquer les identificateurs au-delà d'une certaine longueur. Cette limite dépend des implémentations, mais elle est toujours supérieure à 31 caractères. (Le standard dit que les identificateurs externes, c'est-à-dire ceux qui sont export´es `a l'´edition de lien, peuvent être tronqués à 6 caractères, mais tous les compilateurs modernes distinguent au moins 31 caractères).

4.2 Les mots-clefs

Un certain nombre de mots, appelés mots-clefs, sont réservés pour le langage lui-même et ne peuvent pas être utilisés comme identificateurs. L'ANSI C compte 32 mots clefs :

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

que l'on peut ranger en catégories :

- les spécificateurs de stockage : *auto register static extern typedef*

- les spécificateurs de type :
  char double enum float int long short signed struct union unsigned void
- les qualificateurs de type : *const volatile*
- les instructions de contrôle :
  break case continue default do else for goto if switch while
- divers : *return sizeof*

5. Structure d'un programme C

Une expression est une suite de composants élémentaires syntaxiquement correcte, par
Exemple : *x = 0* ou bien *(i >= 0) && (i < 10) && (p[i] != 0)*

Une instruction est une expression suivie d'un point-virgule. Le point-virgule signifie en quelque sorte "évaluer cette expression".
Plusieurs instructions peuvent être rassemblées pardes accolades { et } pour former une instruction composée ou bloc qui est syntaxiquement équivalent à une instruction. Par exemple :

```
if (x != 0)
{
   z = y / x;
   t = y % x;
}
```

Une instruction composée d'un spécificateur de type et d'une liste d'identificateurs séparés par une virgule est une déclaration. Par exemple :

```
int a;
int b = 1, c;
double x = 2.38e4;
char message[80];
```

En C, toute variable doit faire l'objet d'une déclaration avant d'être utilisée.
Un programme C se présente de la façon suivante :
[directives au préprocesseur]
[déclarations de variables externes]
[fonctions secondaires]

```
main()
{
   déclarations de variables internes
   instructions
}
```

La fonction principale main peut avoir des paramètres formels. On supposera dans un premier temps que la fonction main n'a pas de valeur de retour. Ceci est toléré par le compilateur mais produit un message d'avertissement.
Les fonctions secondaires peuvent être placées indifféremment avant ou après la fonction principale. Une fonction secondaire peut se décrire de la manière suivante :

```
type ma_fonction ( arguments )
{
   déclarations de variables internes
   instructions
}
```

Cette fonction retournera un objet dont le type sera type (à l'aide d'une instruction comme return objet;). Les arguments de la fonction obéissent à une syntaxe voisine de celle des déclarations : on

met en argument de la fonction une suite d'expressions type objet séparées par des virgules. Par exemple, la fonction secondaire suivante calcule le produit de deux entiers :

```
int produit(int a, int b)
{
   int resultat;
   resultat = a * b;
   return(resultat);
}
```

5.1 Les types prédéfinis

5.1.1 Le type caractère

Le mot-clef char désigne un objet de type caractère. Un char peut contenir n'importe quel élément du jeu de caractères de la machine utilisée. La plupart du temps, un objet de type char est codé sur un octet ; c'est l'objet le plus élémentaire en C. Le jeu de caractères utilisé correspond généralement au codage ASCII (sur 7 bits). La plupart des machines utilisent désormais le jeu de caractères ISO-8859 (sur 8 bits).

	déc.	oct.	hex.		déc.	oct.	hex.		déc.	oct.	hex.
	32	40	20	@	64	100	40	‘	96	140	60
!	33	41	21	A	65	101	41	a	97	141	61
”	34	42	22	B	66	102	42	b	98	142	62
#	35	43	23	C	67	103	43	c	99	143	63
$	36	44	24	D	68	104	44	d	100	144	64
%	37	45	25	E	69	105	45	e	101	145	65
&	38	46	26	F	70	106	46	f	102	146	66
’	39	47	27	G	71	107	47	g	103	147	67
(	40	50	28	H	72	110	48	h	104	150	68
)	41	51	29	I	73	111	49	i	105	151	69
*	42	52	2a	J	74	112	4a	j	106	152	6a
+	43	53	2b	K	75	113	4b	k	107	153	6b
,	44	54	2c	L	76	114	4c	l	108	154	6c
-	45	55	2d	M	77	115	4d	m	109	155	6d
.	46	56	2e	N	78	116	4e	n	110	156	6e
/	47	57	2f	O	79	117	4f	o	111	157	6f
0	48	60	30	P	80	120	50	p	112	160	70
1	49	61	31	Q	81	121	51	q	113	161	71
2	50	62	32	R	82	122	52	r	114	162	72
3	51	63	33	S	83	123	53	s	115	163	73
4	52	64	34	T	84	124	54	t	116	164	74
5	53	65	35	U	85	125	55	u	117	165	75
6	54	66	36	V	86	126	56	v	118	166	76
7	55	67	37	W	87	127	57	w	119	167	77
8	56	70	38	X	88	130	58	x	120	170	78
9	57	71	39	Y	89	131	59	y	121	171	79
:	58	72	3a	Z	90	132	5a	z	122	172	7a
;	59	73	3b	[	91	133	5b	{	123	173	7b
<	60	74	3c	\	92	134	5c	|	124	174	7c
=	61	75	3d	]	93	135	5d	}	125	175	7d
>	62	76	3e	^	94	136	5e	~	126	176	7e
?	63	77	3f	_	95	137	5f	DEL	127	177	7f

Codes ASCII des caractères imprimables

Une des particularités du type char en C est qu'il peut être assimilé à un entier : tout objet de type char peut être utilisé dans une expression qui utilise des objets de type entier.

Par exemple, si c est de type char, l'expression c + 1 est valide. Elle désigne le caractère suivant dans le code ASCII. La table de la page 15 donne le code ASCII (en décimal, en octal et en hexadécimal) des caractères imprimables. Ainsi, le programme suivant imprime le caractère 'B'.

```
main()
{
   char c = 'A';
   printf("%c", c + 1);
}
```

Suivant les implémentations, le type char est signé ou non. En cas de doute, il vaut mieux préciser unsigned char ou signed char. Notons que tous les caractères imprimables sont positifs.

5.1.2 Les types entiers

Le mot-clef désignant le type entier est int.
La taille des types n'est que partiellement standardisée
Un objet de type int est représenté par un mot "naturel" de la machine utilisée, 32 bits pour la majorité des processeurs.
Le type int peut être précédé d'un attribut de précision (short ou long) et/ou d'un attribut de représentation (unsigned). Un objet de type short int a au moins la taille d'un char et au plus la taille d'un int. En général, un short int est codé sur 16 bits.
Un objet de type long int a au moins la taille d'un int (64 bits sur un DEC alpha, 32 bits sur un PC Intel).

Commun aux processeurs 32 et 64 bit :

Mode	Type de Base	Taille du type en octets	Nombres de valeurs possibles	Plage de valeurs possible dans le mode
non-signé	unsigned char	1	256	[0 ; 255]
signé	(signed) char	1	256	[-128 ; +127]
non-signé	unsigned short (int)	2	65 536	[0 ; 65 535]
signé	(signed) short (int)	2	65 536	[-32 768 ; +32 767]
non-signé	unsigned (int)	4	4 294 967 296	[0; 4 294 967 295]
signé	(signed) int	4	4 294 967 296	[-2 147 483 648 ; +2 147 483 647]

Spécifiques aux processeurs 32 bits :

Mode	Type de Base	Taille du type en octets	Nombres de valeurs possibles	Plage de valeurs possible dans le mode
non-signé	unsigned long (int)	4	4 294 967 296	[0; 4 294 967 295]
Signé	signed long (int)	4	4 294 967 296	[-2 147 489 648 ; +2 147 489 647]
non-signé	unsigned long long (int)	8	18 446 744 073 509 551 616	[0; 18 446 744 073 509 551 615]
Signé	(signed) long long (int)	8	18 446 744 073 509 551 616	[-9 223 372 036 854 775 808 ; +9 223 372 036 854 775 807]

Spécifiques aux processeurs 64 bits :

Mode	Type de	Taille du type	Nombres de valeurs	Plage de valeurs possible dans le mode

	Base	en octets	possibles	
non-signé	unsigned long (int)	8	18 446 744 073 509 551 616	[0 ; 18 446 744 073 509 551 615]
signé	(signed) long (int)	8	18 446 744 073 509 551 616	[-9 223 372 036 854 775 808 ; +9 223 372 036 854 775 807]

En effet, selon que l'on soit en présence d'un processeur 32 ou 64 bits, certains types ont des tailles et des plages de valeurs variables. C'est pour cette raison que normalement on doit utiliser les types dit "Entiers Fixes" dont nous parlerons plus loin.

Une règle détermine la validité des tailles des types entiers :
[signed|unsigned], char <= short <= int <= long [<= long long]

Plus généralement, les valeurs maximales et minimales des différents types entiers sont définies dans la librairie standard limits.h.
Le mot-clef *sizeof* a pour syntaxe *sizeof(expression)* ou expression est un type ou un objet. Le résultat est un entier égal au nombre d'octets nécessaires pour stocker le type ou l'objet. Par exemple :

```
unsigned short x;
taille = sizeof(unsigned short);
taille = sizeof(x);
```

Dans les deux cas, taille vaudra 4.
Pour obtenir des programmes portables, on s'efforcera de ne jamais présumer de la taille d'un objet de type entier. On utilisera toujours une des constantes de *limits.h* ou le résultat obtenu en appliquant l'opérateur *sizeof*.

5.1.3 Les types flottants

Les types *float*, *double* et *long* double servent à représenter des nombres en virgule flottante. Ils correspondent aux différentes précisions possibles.

Commun aux processeurs 32 et 64 bit :

Type de Base	Taille du type en octets	Taille de l'exposant	Taille de la mantisse	Nombres de valeurs possibles	Plage de valeurs possible
Float	4	8 bits	23 bits	4 294 967 296	(indisponible actuellement)
Double	8	11 bits	52 bits	18 446 744 073 509 551 616	(indisponible actuellement)
long double	12	15 bits	64 bits	79 008 162 513 705 374 134 343 950 336	(indisponible actuellement)

5.2 Les constantes

Une constante est une valeur qui apparaît littéralement dans le code source d'un programme, le type de la constante étant déterminé par la façon dont la constante est écrite. Les constantes peuvent être de 4 types : entier, flottant (nombre réel), caractère, énumération.
Ces constantes vont être utilisées, par exemple, pour initialiser une variable.

5.3 Les opérateurs

5.3.1 L'affectation

En C, l'affectation est un opérateur à part entière. Elle est symbolisée par le signe =. Sa syntaxe est la suivante : *variable = expression*
Le terme de gauche de l'affectation peut être une variable simple, un élément de tableau mais pas une constante. Cette expression a pour effet d'évaluer expression et d'affecter la valeur obtenue à variable. De plus, cette expression possède une valeur, qui est celle expression.
Ainsi, l'expression i = 5 vaut 5.

L'affectation effectue une conversion de type implicite : la valeur de l'expression (terme de droite) est convertie dans le type du terme de gauche. Par exemple, le programme suivant :

```
main()
{
   int i, j = 2;
   float x = 2.5;
   i = j + x;
   x = x + i;
   printf("\n %f \n",x);
}
```

imprime pour x la valeur 6.5 (et non 7), car dans l'instruction i = j + x;, l'expression j + x a été convertie en entier.

5.3.2 Les opérateurs arithmétiques

Les opérateurs arithmétiques classiques sont l'opérateur unaire - (changement de signe) ainsi que les opérateurs binaires :

+ addition
- soustraction
** multiplication*
/ division
% reste de la division (modulo)

Ces opérateurs agissent de la façon attendue sur les entiers comme sur les flottants. Leurs seules spécificités sont les suivantes :

- Contrairement à d'autres langages, le C ne dispose que de la notation / pour désigner à la fois la division entière et la division entre flottants. Si les deux opérandes sont de type entier, l'opérateur / produira une division entière (quotient de la division). Par contre, il délivrera une valeur flottante dès que l'un des opérandes est un flottant. Par exemple :

```
float x;
x = 3 / 2;          //affecte à x la valeur 1. Par contre
x = 3 / 2.;         //affecte à x la valeur 1.5.
```

- L'opérateur % ne s'applique qu'à des opérandes de type entier. Si l'un des deux opérandes est négatif, le signe du reste dépend de l'implémentation, mais il est en général le même que celui du dividende.

  Notons enfin qu'il n'y a pas en C d'opérateur effectuant l'élévation à la puissance. De façon générale, il faut utiliser la fonction *pow(x,y)* de la librairie *math.h* pour calculer xy.

5.3.3 Les opérateurs relationnels

> strictement supérieur
>= supérieur ou égal
< strictement inférieur
<= inf´erieur ou égal
== ´egal
!= différent

Leur syntaxe est expression-1 op expression-2

```
main()
{
        int a = 0;
        int b = 1;
        if (a = b)
        printf("\n a et b sont egaux \n");
        else
        printf("\n a et b sont differents \n");
}
```

imprime à l´´ecran a et b sont égaux !

5.3.4 Les opérateurs logiques booléens

&& et logique
|| ou logique
! négation logique

Comme pour les opérateurs de comparaison, la valeur retournée par ces opérateurs est un int qui vaut 1 si la condition est vraie et 0 sinon.
Dans une expression de type *expression-1 op-1 expression-2 op-2 ...expression-n* l'évaluation se fait de gauche à droite et s'arrête dès que le résultat final est déterminé. Par exemple dans :

```
int i;
int p[10];
if ((i >= 0) && (i <= 9) && !(p[i] == 0))
...
```

la dernière clause ne sera pas évaluée si i n'est pas entre 0 et 9.

5.3.5 Les opérateurs d'affectation composée

Les opérateurs d'affectation composée sont *: += -= *= /= %= &= ^= |= <<= >>=*
Pour tout opérateur *op*, l'expression :

expression-1 op= expression-2

est équivalente à :

expression-1 = expression-1 op expression-2

Toutefois, avec l'affection composée, *expression-1* n'est évaluée qu'une seule fois.

5.3.6 Les opérateurs d'incrémentation et de décrémentation

Les opérateurs d'incrémentation ++ et de décrémentation -- s'utilisent aussi bien en suffixe (i++) qu'en préfixe (++i). Dans les deux cas la variable i sera incrémentée, toutefois dans la notation suffixe la valeur retournée sera l'ancienne valeur de i alors que dans la notation préfixe se sera la nouvelle. Par exemple :

```
int a = 3, b, c;
b = ++a; /* a et b valent 4 */
c = b++; /* c vaut 4 et b vaut 5 */
```

5.3.7 L'opérateur conditionnel ternaire

L'opérateur conditionnel **?** est un opérateur ternaire. Sa syntaxe est la suivante :
condition ? expression-1 : expression-2
Cette expression est égale à expression-1 si condition est satisfaite, et à expression-2 sinon. Par exemple, l'expression : *x >= 0 ? x : -x* correspond à la valeur absolue d'un nombre. De même l'instruction *m = ((a > b) ? a : b);* affecte à m le maximum de a et de b.

5.3.8 L'opérateur de conversion de type

L'opérateur de conversion de type, appelé *cast*, permet de modifier explicitement le type d'un objet.
On écrit : (*type*) *objet*
Par exemple :

```
main()
{
        int i = 3, j = 2;
        printf("%f \n",(float)i/j);
}
```

retourne la valeur 1.5.

5.3.9 L'opérateur adresse

L'opérateur d'adresse & appliqué à une variable retourne l'adresse-mémoire de cette variable.
La syntaxe est *&objet*

5.3.10 Règles de priorité des opérateurs

Le tableau suivant classe les opérateurs par ordres de priorité décroissants. Les opérateurs placés sur une même ligne ont même priorité. Si dans une expression figurent plusieurs opérateurs de même priorité, l'ordre d''evaluation est définie par la flèche de la seconde colonne du tableau. On préférera toutefois mettre des parenthèses en cas de doute...

opérateurs	
() [] -> .	→
! ~ ++ -- -(unaire) (*type*) *(indirection) &(adresse) sizeof	←
* / %	→
+ -(binaire)	→
<< >>	→
< <= > >=	→
== !=	→
&(et bit-à-bit)	→
^	→
|	→
&&	→
||	→
? :	←
= += -= *= /= %= &= ^= |= <<= >>=	←
,	→

Règles de priorité des opérateurs

5.4 Les instructions de branchement conditionnel

On appelle instruction de contrôle toute instruction qui permet de contrôler le fonctionnement d'un programme. Parmi les instructions de contrôle, on distingue les instructions de branchement et les boucles. Les instructions de branchement permettent de déterminer quelles instructions seront exécutées et dans quel ordre.

5.4.1 Branchement conditionnel if---else

La forme la plus générale est celle-ci :

```
if ( expression-1 )
instruction-1
else if ( expression-2 )
instruction-2
...
else if ( expression-n )
```

```
    instruction-n
    else
    instruction-8
```

avec un nombre quelconque de else if (...). Le dernier else est toujours facultatif.
La forme la plus simple est :

```
    if ( expression )
    instruction
```

Chaque instruction peut être un bloc d'instructions.

5.4.2 Branchement multiple switch

Sa forme la plus générale est celle-ci:

```
switch ( expression )
{
    case constante-1 :
    liste d'instructions 1
    break;
    case constante-2 :
    liste d'instructions 2
    break;
    ...
    case constante-n :
    liste d'instructions n
    break;
    default:
    liste d'instructions 8
    break;
}
```

Si la valeur de expression est égale à l'une des constantes , la liste d'instructions correspondant est exécutée. Sinon la liste d'instructions 8 correspondant à default est exécutée. L'instruction default est facultative.

5.5 Les boucles

Les boucles permettent de répéter une série d'instructions tant qu'une certaine condition n'est pas vérifiée.

5.5.1 Boucle while

La syntaxe de while est la suivante :

```
while ( expression )
instruction
```

Tant que expression est vérifiée (i.e. , non nulle), instruction est exécutée. Si expression est nulle au départ, instruction ne sera jamais exécutée. instruction peut évidemment être une instruction composée. Par exemple, le programme suivant imprime les entiers de 1 à 9.

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

5.5.2 Boucle do---while

Il peut arriver que l'on ne veuille exécuter le test de continuation qu'après avoir exécuté l'instruction. Dans ce cas, on utilise la boucle *do---while*. Sa syntaxe est :

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

Ici, instruction sera exécutée tant que expression est non nulle. Cela signifie donc que instruction est toujours exécutée au moins une fois. Par exemple, pour saisir au clavier un entier entre 1 et 10 :

```
int a;
do
{
    printf("\n Entrez un entier entre 1 et 10 : ");
    scanf("%d",&a);
}
while ((a <= 0) || (a > 10));
```

5.5.3 Boucle for

La syntaxe de for est :

```
for (  expr 1  ;  expr 2  ;  expr 3 )
instruction
```

Une version équivalente plus intuitive est :

```
expr 1 ;
while (  expr 2  )
{
    instruction
    expr 3 ;
}
```

Par exemple, pour imprimer tous les entiers de 0 à 9, on écrit :

```
for (i = 0; i < 10; i++)
printf("\n i = %d",i);
```

A la fin de cette boucle, i vaudra 10. Les trois expressions utilisées dans une boucle for peuvent être constituées de plusieurs expressions séparées par des virgules. Cela permet par exemple de faire plusieurs initialisations à la fois. Par exemple, pour calculer la factorielle d'un entier, on peut écrire :

```
int n, i, fact;
for (i = 1, fact = 1; i <= n; i++)
fact *= i;
printf("%d ! = %d \n",n,fact);
```

On peut également insérer l'instruction fact *= i; dans la boucle for ce qui donne :

```
int n, i, fact;
for (i = 1, fact = 1; i <= n; fact *= i, i++);
printf("%d ! = %d \n",n,fact);
```

On évitera toutefois ce type d'acrobaties qui n'apportent rien et rendent le programme difficilement lisible.

5.6 Les instructions de branchement non conditionnel

5.6.1 Branchement non conditionnel break

On a vu le rôle de l'instruction break; au sein d'une instruction de branchement multiple switch. L'instruction break peut, plus généralement, être employée à l'intérieur de n'importe quelle boucle. Elle permet d'interrompre le déroulement de la boucle, et passe à la première instruction qui suit la boucle. En cas de boucles imbriquées, break fait sortir de la boucle la plus interne. Par exemple, le programme suivant :

```
main()
{
    int i;
   for (i = 0; i < 5; i++)
       {
            printf("i = %d\n",i);
           if (i == 3)
               break;
       }
   printf("valeur de i a la sortie de la boucle = %d\n",i);
}
```

imprime à l'écran:

```
i = 0
i = 1
i = 2
i = 3
valeur de i a la sortie de la boucle = 3
```

5.6.2 Branchement non conditionnel continue

L'instruction continue permet de passer directement au tour de boucle suivant, sans exécuter les autres instructions de la boucle. Ainsi le programme

```
main()
{
    int i;
   for (i = 0; i < 5; i++)
       {
            if (i == 3)
               continue;
           printf("i = %d\n",i);
       }
   printf("valeur de i a la sortie de la boucle = %d\n",i);
}
```

imprime

```
i = 0
i = 1
i = 2
i = 4
valeur de i a la sortie de la boucle = 5
```

5.6.3 Branchement non conditionnel goto

L'instruction goto permet d'effectuer un saut jusqu'à l'instruction *étiquette* correspondant.

Elle est à proscrire de tout programme C digne de ce nom.

5.7 Les fonctions d'entrées-sorties classiques

Il s'agit des fonctions de la librairie standard *stdio.h* utilisées avec les unités classiques d'entrées sorties, qui sont respectivement le clavier et l'écran. Sur certains compilateurs, l'appel à la librairie *stdio.h* par la directive au préprocesseur *#include <stdio.h>* n'est pas nécessaire pour utiliser *printf* et *scanf*.

5.7.1 La fonction d'écriture printf

La fonction *printf* est une fonction d'impression formatée, ce qui signifie que les données sont converties selon le format particulier choisi. Sa syntaxe est :

printf(" chaine de contrôle ", expression-1 , ..., expression-n);

La *chaîne de contrôle* contient le texte à afficher et les spécifications de format correspondant à chaque expression de la liste. Les spécifications de format ont pour but d'annoncer le format des données à visualiser. Elles sont introduites par le caractère %, suivi d'un caractère désignant le format d'impression. Les formats d'impression en C sont donnés à la table 1.5.

En plus du caractère donnant le type des données, on peut éventuellement préciser certains paramètres du format d'impression, qui sont spécifiés entre le % et le caractère de conversion dans l'ordre suivant :

- largeur minimale du champ d'impression : %10d spécifie qu'au moins 10 caractères seront réservés pour imprimer l'entier. Par défaut, la donnée sera cadrée à droite du champ. Le signe - avant le format signifie que la donnée sera cadrée à gauche du champ (%-10d).
- Précision : %.12f signifie qu'un fottant sera imprimé avec 12 chiffres après la virgule. De même %10.2f signifie que l'on réserve 12 caractères (incluant le caractère .) pour imprimer le fottant et que 2 d'entre eux sont destinés aux chiffres après la virgule. Lorsque la précision n'est pas spécifiée, elle correspond par défaut à 6 chiffres après la virgule. Pour une chaîne de caractères, la précision correspond au nombre de caractères imprimés :
  %30.4s signifie que l'on réserve un champ de 30 caractères pour imprimer la chaîne mais que seulement les 4 premiers caractères seront imprimés (suivis de 26 blancs).

format	conversion en	écriture
%d	int	décimale signée
%ld	long int	décimale signée
%u	unsigned int	décimale non signée
%lu	unsigned long int	décimale non signée
%o	unsigned int	octale non signée
%lo	unsigned long int	octale non signée
%x	unsigned int	hexadécimale non signée
%lx	unsigned long int	hexadécimale non signée
%f	double	décimale virgule fixe
%lf	long double	décimale virgule fixe
%e	double	décimale notation exponentielle
%le	long double	décimale notation exponentielle
%g	double	décimale, représentation la plus courte parmi %f et %e
%lg	long double	décimale, représentation la plus courte parmi %lf et %le
%c	unsigned char	caractère
%s	char*	chaîne de caractères

Formats d'impression pour la fonction printf

5.7.2 La fonction de saisie scanf

La fonction scanf permet de saisir des données au clavier et de les stocker aux adresses Spécifiées par les arguments de la fonction.

scanf(" chaîne de contrôle ", argument-1 ,..., argument-n)

La *chaîne de contrôle* indique le format dans lequel les données lues sont converties. Elle ne contient pas d'autres caractères (notamment pas de \n). Comme pour *printf*, les conversions de format sont spécifiées par un caractère précédé du signe %. Les formats valides pour la fonction *scanf* diffèrent légèrement de ceux de la fonction *printf*.

Les données à entrer au clavier doivent être séparées par des blancs ou des <RETURN> sauf s'il s'agit de caractères. On peut toutefois fixer le nombre de caractères de la donnée à lire. Par exemple %3s pour une chaîne de 3 caractères, %10d pour un entier qui s'´etend sur 10 chiffres, signe inclus.

Exemple :

```
#include <stdio.h>
main()
{
    int i;
   printf("entrez un entier sous forme hexadecimale i = ");
   scanf("%x",&i);
   printf("i = %d\n",i);
}
```

Si on entre au clavier la valeur 1a, le programme affiche i = 26.

format	type d'objet pointé	représentation de la donnée saisie
%d	int	décimale signée
%hd	short int	décimale signée
%ld	long int	décimale signée
%u	unsigned int	décimale non signée
%hu	unsigned short int	décimale non signée
%lu	unsigned long int	décimale non signée
%o	int	octale
%ho	short int	octale
%lo	long int	octale
%x	int	hexadécimale
%hx	short int	hexadécimale
%lx	long int	hexadécimale
%f	float	flottante virgule fixe
%lf	double	flottante virgule fixe
%Lf	long double	flottante virgule fixe
%e	float	flottante notation exponentielle
%le	double	flottante notation exponentielle
%Le	long double	flottante notation exponentielle
%g	float	flottante virgule fixe ou notation exponentielle
%lg	double	flottante virgule fixe ou notation exponentielle
%Lg	long double	flottante virgule fixe ou notation exponentielle
%c	char	caractère
%s	char*	chaîne de caractères

Formats de saisie pour la fonction `scanf`

5.8 Les conventions d'écriture d'un programme C

Il existe très peu de contraintes dans l'écriture d'un programme C. Toutefois ne prendre aucune précaution aboutirait à des programmes illisibles. Aussi existe-t-il un certain nombre de conventions :

- On n'écrit qu'une seule instruction par ligne : le point virgule d'une instruction ou d'une déclaration est toujours le dernier caractère de la ligne.
- Les instructions sont disposées de telle façon que la structure modulaire du programme soit mise en évidence. En particulier, une accolade ouvrante marquant le début d'un bloc doit être seule sur sa ligne ou placée à la fin d'une ligne. Une accolade fermante est toujours seule sur sa ligne.
- On laisse un blanc :
    - entre les mots-clefs if, while, do, switch et la parenth`ese ouvrante qui suit,
    - après une virgule,
    - de part et d'autre d'un opérateur binaire.
    - On ne met pas de blanc entre un opérateur unaire et son opérande, ni entre les deux caractères d'un opérateur d'affectation composée.
    - Les instructions doivent être indentées afin que toutes les instructions d'un mˆ eme bloc soient alignées.