Bases du langage C

Syntaxe du langage C

Un programme C est un fichier texte qui contient essentiellement :

1. des espaces, saut de lignes, tabulation, commentaires... utiles pour la mise en page

Syntaxe du langage C

Un programme C est un fichier texte qui contient essentiellement :

1. des espaces, saut de lignes, tabulation, commentaires... utiles pour la mise en page
2. des constantes
   - entières : 10
   - flottantes : 10.24
   - caractère simple : 'a'
   - chaîne de caractères : "chaine de caractere"

Syntaxe du langage C

Un programme C est un fichier texte qui contient essentiellement :

1. des espaces, saut de lignes, tabulation, commentaires... utiles pour la mise en page
2. des constantes
   - entières : 10
   - flottantes : 10.24
   - caractère simple : 'a'
   - chaîne de caractères : "chaine de caractere"
3. des opérateurs : +, -, *, /, %...

Syntaxe du langage C

Un programme C est un fichier texte qui contient essentiellement :

1. des espaces, saut de lignes, tabulation, commentaires... utiles pour la mise en page
2. des constantes
   - entières : 10
   - flottantes : 10.24
   - caractère simple : 'a'
   - chaîne de caractères : "chaine de caractere"
3. des opérateurs : +, -, *, /, %...
4. des mots réservés du langage (32 au total)

Syntaxe du langage C

Un programme C est un fichier texte qui contient essentiellement :

1. des espaces, saut de lignes, tabulation, commentaires... utiles pour la mise en page
2. des constantes
   - entières : 10
   - flottantes : 10.24
   - caractère simple : 'a'
   - chaîne de caractères : "chaine de caractere"
3. des opérateurs : +, -, *, /, %...
4. des mots réservés du langage (32 au total)
5. des identifiants (de variable, de fonction, de structure...)

Syntaxe du langage C

Un programme C est un fichier texte qui contient essentiellement :

1. des espaces, saut de lignes, tabulation, commentaires... utiles pour la mise en page
2. des constantes
   - entières : 10
   - flottantes : 10.24
   - caractère simple : 'a'
   - chaîne de caractères : "chaine de caractere"
3. des opérateurs : +, -, *, /, %...
4. des mots réservés du langage (32 au total)
5. des identifiants (de variable, de fonction, de structure...)
6. de la ponctuation : ;, {, }

Mots réservés du langage C

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

Programme type

```
#include <stdio.h>

int main(){
    printf("Hello world\n");
    return 0;
}
```

Les variables

Une variable est un objet constitué par :

- un identifiant

Les variables

Une variable est un objet constitué par :

- un identifiant
- un type qui
  - détermine l'ensemble des valeurs que la variable peut contenir,
  - détermine sa taille mémoire (dépendant de la machine, cf. `sizeof`)

Les variables

Une variable est un objet constitué par :

- un identifiant
- un type qui
  - détermine l'ensemble des valeurs que la variable peut contenir,
  - détermine sa taille mémoire (dépendant de la machine, cf. `sizeof`)
- une adresse en mémoire accessible avec l'opérateur &.

Les variables

Une variable est un objet constitué par :

- un identifiant
- un type qui
  - détermine l'ensemble des valeurs que la variable peut contenir,
  - détermine sa taille mémoire (dépendant de la machine, cf. `sizeof`)
- une adresse en mémoire accessible avec l'opérateur &.
- un domaine de visibilité (par défaut le bloc dans lequel elle est déclarée) et une durée de vie.

Les variables

Une variable est un objet constitué par :

- un identifiant
- un type qui
  - détermine l'ensemble des valeurs que la variable peut contenir,
  - détermine sa taille mémoire (dépendant de la machine, cf. `sizeof`)
- une adresse en mémoire accessible avec l'opérateur &.
- un domaine de visibilité (par défaut le bloc dans lequel elle est déclarée) et une durée de vie.

Toutes ces caractéristiques sont déterminées à la déclaration de la variable.

Exemple

```
#include <stdio.h>

int main(){
    int a,b,c;
    a = 2;
    b = 3;
    /* échange les valeurs de a et b */
    c = b;
    b = a;
    a = c;
    return 0;
}

void autre_fonction(){
    int a,b,c;//même identifiants mais visibilités différentes
    ...
}
```

Types de base

- `void` : utilisé pour les fonctions sans retour et pour l'utilisation des pointeurs.
- `int` $\sim$ 4 octets
- `char` $\sim$ 1 octet : la valeur de la variable est convertie en caractère par l'intermédiaire de la table ASCII.
- `double` $\sim$ 8 octets

Types de base

- `void` : utilisé pour les fonctions sans retour et pour l'utilisation des pointeurs.
- `int` $\sim$ 4 octets
- `char` $\sim$ 1 octet : la valeur de la variable est convertie en caractère par l'intermédiaire de la table ASCII.
- `double` $\sim$ 8 octets

Il n'existe pas de type **booléen**.

Types de base

- `void` : utilisé pour les fonctions sans retour et pour l'utilisation des pointeurs.
- `int` $\sim$ 4 octets
- `char` $\sim$ 1 octet : la valeur de la variable est convertie en caractère par l'intermédiaire de la table ASCII.
- `double` $\sim$ 8 octets

Il n'existe pas de type **booléen**.

Qualificatifs des types :

- `signed, unsigned`
- `short, long`

Code	Caractère	Code	Caractère	Code	Caractère	Code	Caractère	Code	Caractère
0	[car. nul]	69	E	116	t	164	¤	211	Ó
…		70	F	117	u	165	¥	212	Ô
7	[sig. sonore]	71	G	118	v	166	¦	213	Õ
8	[ret. arrière]	72	H	119	w	167	§	214	Ö
9	[tabulation]	73	I	120	x	168	¨	215	×
10	[saut ligne]	74	J	121	y	169	©	216	Ø
11	[tab. vert.]	75	K	122	z	170	ª	217	Ù
12	[saut page]	76	L	123	{	171	«	218	Ú
13	[ret. chariot]	77	M	124	|	172	¬	219	Û
...		78	N	125	}	173	-	220	Ü
32	[espace]	79	O	126	~	174	®	221	Ý
33	!	80	P	…		175	¯	222	Þ
34	"	81	Q	128	€	176	°	223	ß
35	#	82	R	…		177	±	224	à
36	$	83	S	130	‚	178	²	225	á
37	%	84	T	131	*f*	179	³	226	â
38	&	85	U	132	„	180	´	227	ã
39	’	86	V	133	…	181	µ	228	ä
40	(	87	W	134	†	182	¶	229	å
41	)	88	X	135	‡	183	·	230	æ
42	*	89	Y	136	ˆ	184	¸	231	ç
43	+	90	Z	137	‰	185	¹	232	è
44	,	91	[	138	Š	186	º	233	é
45	-	92	\	139	‹	187	»	234	ê
46	.	93	]	140	Œ	188	¼	235	ë
47	/	94	^	…		189	½	236	ì
48	0	95	_	142	Ž	190	¾	237	í
49	1	96	`	…		191	¿	238	î
50	2	97	a	145	‘	192	À	239	ï
51	3	98	b	146	’	193	Á	240	ð
52	4	99	c	147	“	194	Â	241	ñ
53	5	100	d	148	”	195	Ã	242	ò
54	6	101	e	149	•	196	Ä	243	ó
55	7	102	f	150	–	197	Å	244	ô
56	8	103	g	151	—	198	Æ	245	õ
57	9	104	h	152	˜	199	Ç	246	ö

Opérateurs et expressions

Les propriétés des opérateurs :

- nombre d'opérandes,
- priorité,
- associativité : $1.0/(2.0/3.0) \neq (1.0/2.0)/3.0$.

Opérateurs et expressions

Les propriétés des opérateurs :

- nombre d'opérandes,
- priorité,
- associativité : $1.0/(2.0/3.0) \neq (1.0/2.0)/3.0$.

Une **expression** est une suite d'opérateurs et d'opérandes :

- une expression a une valeur
- elle peut avoir un effet de bord (par exemple l'affectation).

Opérateurs et expressions

Les propriétés des opérateurs :

- nombre d'opérandes,
- priorité,
- associativité : $1.0/(2.0/3.0) \neq (1.0/2.0)/3.0$.

Une **expression** est une suite d'opérateurs et d'opérandes :

- une expression a une valeur
- elle peut avoir un effet de bord (par exemple l'affectation).

Conseil : en dehors des opérateurs d'arithmétique usuels, il est préférable de parenthéser les expressions.

Opérateurs et expressions

Les propriétés des opérateurs :

- nombre d'opérandes,
- priorité,
- associativité : $1.0/(2.0/3.0) \neq (1.0/2.0)/3.0$.

Une **expression** est une suite d'opérateurs et d'opérandes :

- une expression a une valeur
- elle peut avoir un effet de bord (par exemple l'affectation).

Conseil : en dehors des opérateurs d'arithmétique usuels, il est préférable de parenthéser les expressions.

Attention : a part quelques exceptions, l'ordre d'évaluation des opérandes n'est pas spécifié par le langage.

```
t[i] = f(i) // fonction qui modifie i
```

Catégorie d'opérateurs	Symbole	Arité	Associativité
fonction, tableau, membre de structure, pointeur sur un membre de structure	`() [] . ->`	2	G ⇒ D
opérateurs unaires	`+ - ++ -- ! ~` `* & sizeof (type)`	1	D ⇒ G
multiplication, division, modulo	`* / %`	2	G ⇒ D
addition, soustraction	`+ -`	2	G ⇒ D
opérateurs binaires de décalage	`<< >>`	2	G ⇒ D
opérateurs relationnels	`< <= > >=`	2	G ⇒ D
opérateurs de comparaison	`== !=`	2	G ⇒ D
et binaire	`&`	2	G ⇒ D
ou exclusif binaire	`^`	2	G ⇒ D
ou binaire	`|`	2	G ⇒ D
et logique	`&&`	2	G ⇒ D
ou logique	`||`	2	G ⇒ D
opérateur conditionnel	`?:`	3	D ⇒ G
opérateurs d'affectation	`= += -= *= /= %=` `&= ^= |= <<= >>=`	2	D ⇒ G
opérateur virgule	`,`	2	G ⇒ D

Conversion implicite pour les opérateurs binaires

Pas de problèmes si les deux opérandes ont le même type valide pour l'opérateur.
Si c'est une opération arithmétique, le type du résultat est identique.

Conversion implicite pour les opérateurs binaires

Pas de problèmes si les deux opérandes ont le même type valide pour l'opérateur.
Si c'est une opération arithmétique, le type du résultat est identique.

Sinon l'opérande qui a le type le plus à gauche dans la liste suivante est convertie dans le type de l'autre opérande sans perte d'information.
C'est aussi le type du résultat.

```
char -> int -> unsigned int -> double
```

Conversion implicite pour les opérateurs binaires

Pas de problèmes si les deux opérandes ont le même type valide pour l'opérateur.
Si c'est une opération arithmétique, le type du résultat est identique.

Sinon l'opérande qui a le type le plus à gauche dans la liste suivante est convertie dans le type de l'autre opérande sans perte d'information.
C'est aussi le type du résultat.

```
char -> int -> unsigned int -> double
```

Exemple : 5/2 vaut 2 alors que 5.0/2 vaut 2.5

Changement du type d'une variable

Il est parfois possible de convertir la valeur d'une variable dans un autre type (cast) de la manière suivante :

```
(type) variable
(int)  nombre_decimal //partie décimale tronquée
```

Changement du type d'une variable

Il est parfois possible de convertir la valeur d'une variable dans un autre type (cast) de la manière suivante :

```
(type) variable
(int)  nombre_decimal //partie décimale tronquée
```

Exemple :

```
int a = 2;
int b = 3;
b / a; /* division entière */
(double)b / (double)a; /* division de doubles */
b * 1.0 / a; /* idem */
```

Piège

```
int main(){
  if ((unsigned int) 10 < -1)
    printf("Bizarre, bizarre ...\n");
  return 0;
}
```

Opérateur d'affectation

Pour affecter une valeur à une variable :

```
variable = valeur
```

Opérateur d'affectation

Pour affecter une valeur à une variable :

```
variable = valeur
```

On ne peut pas écrire `1 = 5`

```
erreur: lvalue required as left operand of assignment
```

Une `lvalue` est une valeur associée à un type et une adresse.

Opérateur d'affectation

Pour affecter une valeur à une variable :

```
variable = valeur
```

On ne peut pas écrire `1 = 5`

```
erreur: lvalue required as left operand of assignment
```

Une `lvalue` est une valeur associée à un type et une adresse.

L'opérateur d'affectation a deux effets :

- effet de bord : elle affecte la valeur à la variable,
- résultat de l'opérateur : elle renvoie la valeur affectée.

Opérateur d'affectation

Pour affecter une valeur à une variable :

```
variable = valeur
```

On ne peut pas écrire `1 = 5`

```
erreur: lvalue required as left operand of assignment
```

Une `lvalue` est une valeur associée à un type et une adresse.

L'opérateur d'affectation a deux effets :

- effet de bord : elle affecte la valeur à la variable,
- résultat de l'opérateur : elle renvoie la valeur affectée.

```
a3 = (a2 = (a1 = 1) + 1) + 1;
```

Opérateur d'affectation

Pour affecter une valeur à une variable :

```
variable = valeur
```

On ne peut pas écrire `1 = 5`

```
erreur: lvalue required as left operand of assignment
```

Une `lvalue` est une valeur associée à un type et une adresse.

L'opérateur d'affectation a deux effets :

- effet de bord : elle affecte la valeur à la variable,
- résultat de l'opérateur : elle renvoie la valeur affectée.

```
a3 = (a2 = (a1 = 1) + 1) + 1;
```

Le type de `valeur` est éventuellement converti dans le type de `variable`.

Opérateurs d'incrémentation

Opérateurs unaires `++` et `--` pour des opérandes de type `int`.

Opérateurs d'incrémentation

Opérateurs unaires `++` et `--` pour des opérandes de type `int`.

Effet de bord de `++i` et `i++` : incrémente la variable `i` d'une unité.

Opérateurs d'incrémentation

Opérateurs unaires `++` et `--` pour des opérandes de type `int`.

Effet de bord de `++i` et `i++` : incrémente la variable `i` d'une unité.

Différence entre `++i` et `i++` : la valeur renvoyée est la valeur de `i` respectivement après et avant l'incrémentation.

Opérateurs d'incrémentation

Opérateurs unaires `++` et `--` pour des opérandes de type `int`.

Effet de bord de `++i` et `i++` : incrémente la variable `i` d'une unité.

Différence entre `++i` et `i++` : la valeur renvoyée est la valeur de `i` respectivement après et avant l'incrémentation.

Laquelle des expressions est équivalente à `i=i+1` ?

Opérateurs de comparaison

Les opérateurs de comparaison sont

`<`, `<=`, `>`, `>=`, `==`, `!=`

Opérateurs de comparaison

Les opérateurs de comparaison sont

`<`, `<=`, `>`, `>=`, `==`, `!=`

La valeur du resultat est

- 0 si FAUX,
- 1 si VRAI.

Opérateurs de logique (opérandes int ou double)

ET logique :

```
expression1 && expression2
```

- si `expression1` (de type `int`) vaut 0, renvoie 0,
- sinon si `expression2` vaut 0, renvoie 0,
- sinon renvoie 1.

Opérateurs de logique (opérandes int ou double)

ET logique :

```
expression1 && expression2
```

- si `expression1` (de type `int`) vaut 0, renvoie 0,
- sinon si `expression2` vaut 0, renvoie 0,
- sinon renvoie 1.

OU logique :

```
expression1 || expression2
```

- si `expression1` différent de 0, renvoie 1,
- sinon si `expression2` vaut 0, renvoie 0,
- sinon renvoie 1.

Opérateurs de logique (opérandes int ou double)

ET logique :

```
expression1 && expression2
```

- si `expression1` (de type `int`) vaut 0, renvoie 0,
- sinon si `expression2` vaut 0, renvoie 0,
- sinon renvoie 1.

OU logique :

```
expression1 || expression2
```

- si `expression1` différent de 0, renvoie 1,
- sinon si `expression2` vaut 0, renvoie 0,
- sinon renvoie 1.

NON logique :

```
! expression
```

- si `expression` vaut 0, renvoie 1,
- sinon renvoie 0.

Opérateurs de logique

Remarque 1 :
`!(expr1 || expr2)` est identique à `(! expr1) && (! expr2)` .
Loi de Morgan

Opérateurs de logique

Remarque 1 :
`!(expr1 || expr2)` est identique à `(! expr1) && (! expr2)` .
Loi de Morgan

Remarque 2 :
Soit `arreter(...)` une fonction qui vaut 0 ou 1 (par ex. teste une condition d'arrêt de boucle).

```
while(! arreter(...)){
    ...
}
```

Opérateur conditionnel ternaire

```
expression1 ? expression2 : expression3
```

- si `expression1` est non nulle, le résultat est la valeur de `expression2`.
- Sinon le résultat est la valeur de `expression3`.

Opérateur conditionnel ternaire

```
expression1 ? expression2 : expression3
```

- si `expression1` est non nulle, le résultat est la valeur de `expression2`.
- Sinon le résultat est la valeur de `expression3`.

```
max = ( a > b ? a : b );
```

Opérateur conditionnel ternaire

```
expression1 ? expression2 : expression3
```

- si `expression1` est non nulle, le résultat est la valeur de `expression2`.
- Sinon le résultat est la valeur de `expression3`.

```
max = ( a > b ? a : b );
```

```
u = ( u%2 == 0 ? u / 2 : 3*u + 1 );
```

Donner la valeur des affectations

```
i = 3; a = i++;
```

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

```
a = !(i == 3) ;
```

```
lettre=(c <= 'z' && c >= 'a' ? c : 'a' + c - 'A')
                        //si c vaut 'e' et c vaut 'M'
```

```
a = -1.0 / 2 * 5;
```

```
a = -1 / 2 * 5.0;
```