Programmation Avancée

Cours 7 : Les chaînes de caractères

Simon Forest

18 mars 2021

Caractères et chaînes en C

En C, vous avez sûrement appris que :

- `char` est le type des caractères
- `char*` est le type des chaînes de caractères
- `strlen` renvoie la taille d'une chaîne de caractères
- *etc.*

Caractères et chaînes en C

En fait, c'est un peu plus compliqué que cela...

```
char c = 'é';
```

```
warning: multi-character character constant
warning: overflow in implicit constant conversion
```

```
printf("%ld",strlen("été"));
```

```
5
```

Caractères et chaînes en C

`char` est avant tout le type des **octets**. Il contient au maximum $2^8 = 256$ valeurs.

Dans le monde, on trouve bien plus que 256 caractères différents. Logiquement, certains ne tiennent pas sur un `char`, comme

```
é è œ ß « »
```

Autres langages

Dans les langages modernes, il y a une distinction claire entre

- les caractères
- les chaînes de caractères
- les octets
- les tableaux d'octets

Exemple en Java :

```
char c      = 'œ';
String str = "« Cet été-là ! », me dit-il."

byte b      = 0x43;
byte[] tab = { 0x22, 0x25, 0x34 };
```

En C

En C, tout est mélangé avec le type `char` :

```
char c = 'a';
char *ptr = "Bonne journée !";

char c = 0x42;
char tab[3] = {0x11, 0x22, 0x33};
```

La table ASCII

Même si les `char` ne permettent pas de représenter tous les caractères, ils permettent d'en représenter un certain nombre : ceux de la table ASCII.

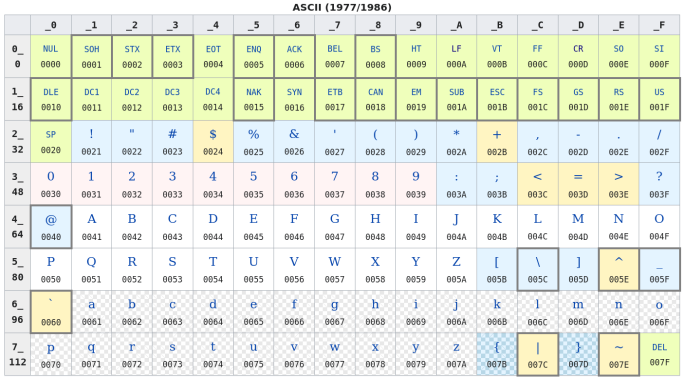
Cette table précise comment interpréter les `char` de valeurs entre 0 et 127.

La table ASCII

ASCII (1977/1986)

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_ 0	NUL 0000	SOH 0001	STX 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
1_ 16	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
2_ 32	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3_ 48	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4_ 64	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5_ 80	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6_ 96	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7_ 112	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	DEL 007F

La table ASCII



ASCII (1977/1986)

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_ 0	NUL 0000	SOH 0001	STX 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
1_ 16	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
2_ 32	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3_ 48	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4_ 64	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5_ 80	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6_ 96	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7_ 112	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	DEL 007F

Exemples :

- le caractère `'!'` représente la valeur `0x21` (33 en décimal)

La table ASCII

ASCII (1977/1986)

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_ 0	NUL 0000	SOH 0001	STX 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
1_ 16	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
2_ 32	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3_ 48	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4_ 64	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5_ 80	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6_ 96	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7_ 112	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	DEL 007F

Exemples :

- le caractère `'B'` représente la valeur `0x42` (66 en décimal)

La table ASCII

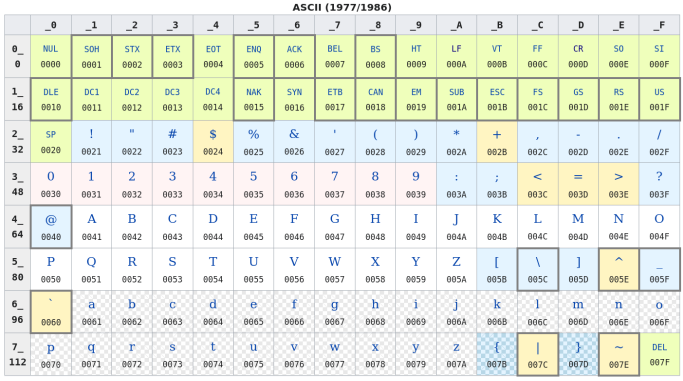
ASCII (1977/1986)

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_ 0	NUL 0000	SOH 0001	STX 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
1_ 16	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
2_ 32	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3_ 48	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4_ 64	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5_ 80	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6_ 96	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7_ 112	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	DEL 007F

Exemples :

- le caractère `'0'` représente la valeur `0x30` (48 en décimal)

La table ASCII



ASCII (1977/1986)

	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0_ 0	NUL 0000	SOH 0001	STX 0002	ETX 0003	EOT 0004	ENQ 0005	ACK 0006	BEL 0007	BS 0008	HT 0009	LF 000A	VT 000B	FF 000C	CR 000D	SO 000E	SI 000F
1_ 16	DLE 0010	DC1 0011	DC2 0012	DC3 0013	DC4 0014	NAK 0015	SYN 0016	ETB 0017	CAN 0018	EM 0019	SUB 001A	ESC 001B	FS 001C	GS 001D	RS 001E	US 001F
2_ 32	SP 0020	! 0021	" 0022	# 0023	$ 0024	% 0025	& 0026	' 0027	(0028	) 0029	* 002A	+ 002B	, 002C	- 002D	. 002E	/ 002F
3_ 48	0 0030	1 0031	2 0032	3 0033	4 0034	5 0035	6 0036	7 0037	8 0038	9 0039	: 003A	; 003B	< 003C	= 003D	> 003E	? 003F
4_ 64	@ 0040	A 0041	B 0042	C 0043	D 0044	E 0045	F 0046	G 0047	H 0048	I 0049	J 004A	K 004B	L 004C	M 004D	N 004E	O 004F
5_ 80	P 0050	Q 0051	R 0052	S 0053	T 0054	U 0055	V 0056	W 0057	X 0058	Y 0059	Z 005A	[005B	\ 005C	] 005D	^ 005E	_ 005F
6_ 96	` 0060	a 0061	b 0062	c 0063	d 0064	e 0065	f 0066	g 0067	h 0068	i 0069	j 006A	k 006B	l 006C	m 006D	n 006E	o 006F
7_ 112	p 0070	q 0071	r 0072	s 0073	t 0074	u 0075	v 0076	w 0077	x 0078	y 0079	z 007A	{ 007B	| 007C	} 007D	~ 007E	DEL 007F

Exemples :

- ... **en particulier**, `'0' != 0` ! De même, `'1' != 1`, `'2' != 2`, *etc.*

La table ASCII

On voit aussi qu'on a des **plages continues** de valeurs du même type :

- les caractères `'0'`, `'1'`, ..., `'9'` représentent les valeurs `0x30`, `0x31`, ..., `0x39`
- les caractères `'A'`, `'B'`, ..., `'Z'` représentent les valeurs `0x41`, `0x42`, ..., `0x5A`
- les caractères `'a'`, `'b'`, ..., `'z'` représentent les valeurs `0x61`, `0x62`, ..., `0x7A`

Cela est pratique pour faire un certain nombre d'opérations sur les caractères.

Exemple

Comment tester si un `char` est un chiffre ?

Exemple

Comment tester si un `char` est un chiffre ?

Mauvaise réponse :

```
int est_chiffre(char c) {
  return 0 <= c <= 9;
}
```

Exemple

Comment tester si un `char` est un chiffre ?

Une autre mauvaise réponse :

```
int est_chiffre(char c) {
  return '0' <= c <= '9';
}
```

Exemple

Comment tester si un `char` est un chiffre ?

Bonne réponse :

```
int est_chiffre(char c) {
  return '0' <= c && c <= '9';
}
```

Exemple

Comment passer d'un `char` qui est un chiffre à la valeur de ce chiffre ?

Exemple

Comment passer d'un `char` qui est un chiffre à la valeur de ce chiffre ?

- `'0'` $\rightsquigarrow$ `0`
- `'4'` $\rightsquigarrow$ `4`
- *etc.*

Exemple

Comment passer d'un `char` qui est un chiffre à la valeur de ce chiffre ?

Réponse :

```
int chiffre_vers_int(char c) {
  return c - '0';
}
```

Caractères spéciaux

La table ASCII ne donne pas de valeurs aux caractères « spéciaux » (c-à-d absents de l'anglais) comme é, è, œ, *etc.*

Aussi, l'ASCII a attribué un caractère seulement aux 128 premières valeurs des `char`.
On peut donc encore en attribuer 128 pour représenter d'autres caractères.

Deux solutions ont été implémentées historiquement pour gérer les caractères spéciaux :

- garder un encodage des caractères sur un octet, en donnant une interprétation aux 128 valeurs restantes
- s'autoriser à utiliser davantage d'octets pour coder les caractères

ISO 8859

L'encodage en ISO 8859

- n'utilise toujours que 1 octet pour représenter les caractères
- donne la même interprétation que l'ASCII pour les 128 premières valeurs de `char`
- représente les caractères spécifiques des langues non-anglaises dans les 128 autres valeurs de `char`
- a différentes versions, couvrant différents groupes de langues : ISO 8859-1, ISO 8859-2, ISO 8859-3, *etc.*

ISO 8859

L'encodage en ISO 8859

- n'utilise toujours que 1 octet pour représenter les caractères
- donne la même interprétation que l'ASCII pour les 128 premières valeurs de `char`
- représente les caractères spécifiques des langues non-anglaises dans les 128 autres valeurs de `char`
- a différentes versions, couvrant différents groupes de langues : ISO 8859-1, ISO 8859-2, ISO 8859-3, *etc.*

Pour le français, on utilise

- soit ISO 8859-1 (sans œ, Œ)
- soit ISO 8859-15 (avec œ, Œ)

La table ISO 8859-15

ISO 8859-15																
	x0	**x1**	**x2**	**x3**	**x4**	**x5**	**x6**	**x7**	**x8**	**x9**	**xA**	**xB**	**xC**	**xD**	**xE**	**xF**
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
8x	PAD	HOP	BPH	NBH	IND	NEL	SSA	ESA	HTS	HTJ	VTS	PLD	PLU	RI	SS2	SS3
9x	DCS	PU1	PU2	STS	CCH	MW	SPA	EPA	SOS	SGCI	SCI	CSI	ST	OSC	PM	APC
Ax	NBSP	¡	¢	£	€	¥	Š	§	š	©	ª	«	¬		®	¯
Bx	°	±	²	³	Ž	µ	¶	·	ž	¹	º	»	Œ	œ	Ÿ	¿
Cx	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
Dx	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
Ex	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
Fx	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

La table ISO 8859-7 (pour le grec)

ISO/CEI 8859-7:2003															
	x0	**x1**	**x2**	**x3**	**x4**	**x5**	**x6**	**x7**	**x8**	**x9**	**xA**	**xB**	**xC**	**xD**	**xE**	**xF**
0x	*Inutilisé*															
1x																
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
8x	*Inutilisé*															
9x																
Ax	NBSP	‘	’	£	€	₯	¦	§	¨	©	ͺ	«	¬	SHY		―
Bx	°	±	²	³	΄	΅	Ά	·	Έ	Ή	Ί	»	Ό	½	Ύ	Ώ
Cx	ΐ	Α	Β	Γ	Δ	Ε	Ζ	Η	Θ	Ι	Κ	Λ	Μ	Ν	Ξ	Ο
Dx	Π	Ρ		Σ	Τ	Υ	Φ	Χ	Ψ	Ω	Ϊ	Ϋ	ά	έ	ή	ί
Ex	ΰ	α	β	γ	δ	ε	ζ	η	θ	ι	κ	λ	μ	ν	ξ	ο
Fx	π	ρ	ς	σ	τ	υ	φ	χ	ψ	ω	ϊ	ϋ	ό	ύ	ώ	

Problème

ISO 8859 permet d'écrire correctement dans une langue qui n'est pas l'anglais ...

Problème

ISO 8859 permet d'écrire correctement dans une langue qui n'est pas l'anglais ...

... mais **pas plusieurs à la fois**.

Problème

ISO 8859 permet d'écrire correctement dans une langue qui n'est pas l'anglais ...

... mais **pas plusieurs à la fois**.

C'est souvent gênant, car on a souvent besoin d'utiliser des caractères d'une autre langue pour

- faire des citations :

  Comme on dit en russe, « Видит око, да зуб неймёт ».

- faire des mathématiques :

  $$\textit{Si } \alpha = \frac{\pi}{2}\textit{, alors } \sin(\alpha) = 1.$$

- *etc.*

Unicode

Le **standard Unicode** a été introduit afin de pouvoir écrire des documents dans **n'importe quelle langue**.

Il consiste en :

- une table avec tous les caractères existants,
- une façon d'accéder aux éléments de la table par des séquences d'octets : UTF-8, UTF-16, UTF-32.

UTF-8

UTF-8 permet d'accéder à tous les caractères décrits dans Unicode.

Chaque caractères est représenté par UTF-8 par **un nombre variable d'octets** : entre 1 et 4.

Cet encodage est compatible avec ASCII : les 128 caractères définis par ASCII sont représentés de la même façon par UTF-8.

C'est le **format standard** pour toutes les langues occidentales.

Exemples

- le caractère « B » est encodé par 1 octet : `0x42` (comme en ASCII)
- le caractère « h » est encodé par 1 octet : `0x68` (comme en ASCII)
- *etc.*

Exemples

- le caractère « è » est encodé par 2 octets : `0xC3,0xA8`
- le caractère « œ » est encodé par 2 octets : `0xC5,0x93`
- le caractère « α » est encodé par 2 octets : `0xC9,0x91`

Représenter les chaînes en C : en théorie

Les chaînes de caractères sont des séquences finies de caractères.

Représenter les chaînes en C : en théorie

Les chaînes de caractères sont des séquences finies de caractères.

Les caractères peuvent tous être représentés par des éléments de Unicode.

Représenter les chaînes en C : en théorie

Les chaînes de caractères sont des séquences finies de caractères.

Les caractères peuvent tous être représentés par des éléments de Unicode.

Ainsi, on pourrait fidèlement représenter les caractères et les chaînes en C par :

```c
typedef struct { /* ... */ } unicode_char;

typedef struct {
  int taille;
  unicode_char *tab;
} unicode_string;
```

Représenter les chaînes en C : en théorie

Les chaînes de caractères sont des séquences finies de caractères.

Les caractères peuvent tous être représentés par des éléments de Unicode.

Ainsi, on pourrait fidèlement représenter les caractères et les chaînes en C par :

```c
typedef struct { /* ... */ } unicode_char;

typedef struct {
  int taille;
  unicode_char *tab;
} unicode_string;
```

Cependant, ce n'est pas ce qui est effectivement utilisé. Causes : héritage historique de « 1 caractère = 1 octet », et soucis de performance.

Représenter les chaînes en C : en pratique

À la place, le C représente les chaînes par des `char*`.

Représenter les chaînes en C : en pratique

À la place, le C représente les chaînes par des `char*`.

Mais, sans indication supplémentaire, comment savoir où s'arrête une chaîne `char* str` ?

Représenter les chaînes en C : en pratique

À la place, le C représente les chaînes par des `char*`.

Mais, sans indication supplémentaire, comment savoir où s'arrête une chaîne `char* str` ?

Pour répondre à cette question **efficacement**, le C demande que :

- les chaînes manipulées ne contiennent pas le **caractère nul** `'\0'` (encodé par 0) ;
- une chaîne de n octets sera en fait représenté par $n+1$ octets, où le dernier octet sera `'\0'`.

On parle de C-**chaîne** (ou *C-string*)

Représenter les chaînes en C : en pratique

À la place, le C représente les chaînes par des `char*`.

Mais, sans indication supplémentaire, comment savoir où s'arrête une chaîne `char* str` ?

Pour répondre à cette question **efficacement**, le C demande que :

- les chaînes manipulées ne contiennent pas le **caractère nul** `'\0'` (encodé par 0) ;
- une chaîne de n octets sera en fait représenté par $n+1$ octets, où le dernier octet sera `'\0'`.

On parle de C-**chaîne** (ou *C-string*)

En acceptant de s'interdire `'\0'`, on encode les chaînes avec un simple pointeur sans un `int` supplémentaire spécifiant la taille : **efficacité**.

Déclaration de chaînes

Les chaînes de caractères se définissent dans le code par des séquences de caractères entre `"` ... `"`.

Déclaration de chaînes

Les chaînes de caractères se définissent dans le code par des séquences de caractères entre `"` ... `"`.

On peut écrire certains caractères spéciaux en utilisant `\` suivi d'une lettre :

- `'\n'` : caractère « saut de ligne »
- `'\r'` : caractère « retour début de ligne »
- `'\t'` : caractère « tabulation »
- *etc.*

Déclaration de chaînes

Les chaînes de caractères se définissent dans le code par des séquences de caractères entre `"` ... `"`.

On peut écrire certains caractères spéciaux en utilisant `\` suivi d'une lettre :

- `'\n'` : caractère « saut de ligne »
- `'\r'` : caractère « retour début de ligne »
- `'\t'` : caractère « tabulation »
- *etc.*

On peut aussi donner directement le code d'un octet sous la forme `\xxx` où `xxx` est une décomposition en base 8 de l'octet :

- `'\0'` est le **caractère nul**
- `'\101'` est l'octet de valeur 65 qui correspond à `'A'` en ASCII
- *etc.*

Exemple

Qu'affiche ce programme ?

```
char *str = "\101lbert\nSophi\145\0Germain";
printf("%s",str);
```

Exemple

Qu'affiche ce programme ?

```
char *str = "\101lbert\nSophi\145\0Germain";
printf("%s",str);
```

Réponse :

```
Albert
Sophie
```

Une fonction sur les chaînes : `strlen`

`strlen(char *str)` : renvoie la taille **en octets** (pas en caractères !) d'une chaîne.

Une fonction sur les chaînes : `strlen`

`strlen(char *str)` : renvoie la taille **en octets** (pas en caractères !) d'une chaîne.

```
int strlen(char *str){
  int i = 0;
  while(str[i] != '\0')
    i++;
  return i;
}
```

Une fonction sur les chaînes : `strlen`

`strlen(char *str)` : renvoie la taille **en octets** (pas en caractères !) d'une chaîne.

```
int strlen(char *str){
  int i = 0;
  while(str[i] != '\0')
    i++;
  return i;
}
```

On comprend alors pourquoi `strlen("été") == 5`.

Fonctions `str...`

En fait, toutes les fonctions en `str...` travaillent avec des C-chaînes.

On peut y accéder après avoir fait `#include <string.h>`.

Fonctions `str...`

```
char * strcpy ( char * dest, const char * src );
```

Copie la chaîne contenue à `src` dans `dest`. La copie s'arrête quand `'\0'` est rencontré.

Attention, il faut que la mémoire pointée par `dest` soit assez grande pour stocker `src`. Sinon, corruption mémoire ou `segfault`.

```
char dest[100] = "Bonjour";
char *src = "Guten Tag";
printf("%s\n",dest);
strcpy(dest,src); // ok, car "Guten Tag" ne nécessite que 10 octets
printf("%s\n",dest);
```

```
Bonjour
Guten Tag
```

Fonctions `str...`

```
char * strcat ( char * dest, const char * src );
```

Ajoute la chaîne contenue à `src` à la fin de `dest`. L'ajout s'arrête quand `'\0'` est rencontré.

Attention, il faut que la mémoire pointée par `dest` soit assez grande pour stocker la chaîne étendue. Sinon, corruption mémoire ou `segfault`.

```
char dest[100] = "Je pense ";
char *src = "donc je suis.";
strcat(dest,src); // ok, car "Je pense donc je suis." < 100
printf("%s\n",dest);
```

```
Je pense donc je suis.
```

Fonctions `str...`

```
int strcmp ( const char * str1, const char * str2 );
```

Compare deux chaînes de caractères. La comparaison s'arrête quand `'\0'` est rencontré, voire avant.

```
char *str1 = "aaa";
char *str2 = "bbb";
printf("%d\n",strcmp(str1,str2));
printf("%d\n",strcmp(str2,str1));
printf("%d\n",strcmp(str1,str1));
```

```
-1
1
0
```

Fonctions `str...`

```
const char * strstr ( const char * str1, const char * str2 );
```

Trouve la première occurence de `str2` dans `str1`. Renvoie `NULL` si la chaîne n'est pas trouvée.

```
char *str1 = "Les chaussettes de l'archiduchesse";
char *str2 = "aus";
printf("%s",strstr(str1,str2));
```

```
aussettes de l'archiduchesse
```

Fonctions `str...`

```
char* strdup(const char *str);
```

Fournit une copie de la C-chaîne `str`.

```
char *str1 = "Baba";
char *str2 = strdup(str1);
str2[1] = str2[3] = 'i';
printf("str1: %s\n",str1);
printf("str2: %s\n",str2);
```

```
str1: Baba
str2: Bibi
```

Entrées / sorties

Plusieurs fonctions de `#include <stdio.h>` travaillent avec des C-chaînes.

```
printf("%s",str);
```

Écrit le contenu de `str` jusqu'à rencontrer `'\0'`.

Entrées / sorties

Plusieurs fonctions de `#include <stdio.h>` travaillent avec des C-chaînes.

```
printf("%s",str);
```

Écrit le contenu de `str` jusqu'à rencontrer `'\0'`.

```
scanf("%s",str);
```

Récupère un **mot** sur l'entrée et le met dans `str`. Termine la chaîne lue par `'\0'`.
Attention, il faut que la mémoire disponible à `str` soit assez longue.

Entrées / sorties

Plusieurs fonctions de `#include <stdio.h>` travaillent avec des C-chaînes.

```
scanf("%[^\n]",str);
```

Récupère une **ligne** sur l'entrée et la met dans `str`. Termine la chaîne lue par `'\0'`.
Attention, il faut que la mémoire disponible à `str` soit assez longue.

Entrées / sorties

Plusieurs fonctions de `#include <stdio.h>` travaillent avec des C-chaînes.

```
scanf("%[^\n]",str);
```

Récupère une **ligne** sur l'entrée et la met dans `str`. Termine la chaîne lue par `'\0'`.
Attention, il faut que la mémoire disponible à `str` soit assez longue.

```
char* gets(char * str);
```

Récupère une ligne entière de l'entrée et la met dans `str`. Renvoie `NULL` si aucun caractère n'a pu être lu.
Attention, il faut que la mémoire disponible à `str` soit assez longue.

Entrées / sorties

Plusieurs fonctions de `#include <stdio.h>` travaillent avec des C-chaînes.

```
ssize_t getline (char **ptr, size_t *taille, FILE *f)
```

Permet de lire **de façon sûre** une ligne entière d'un fichier `f`. La valeur retournée est le nombre de caractères lus.

Utilisation simple :

```
char *str = NULL;
size_t taille = 0;
int r = getline(&ptr,&taille,stdin);
if(r > 0){ /* une ligne a été lue */ }
```

Fonctions `mem...`

Rappel : `char` est aussi le type des octets, et `char*` le type des blocs d'octets.

Le *header* `#include <string.h>` propose aussi des fonctions pour manipuler des blocs d'octets, sans les interpréter en C-chaînes.

Ces fonctions commencent en `mem...`.

Fonctions `mem...`

```
char * memcpy (char * dest, const char * src, size_t taille );
```

Copie `taille` octets à partir de `src` vers `dest`.

Attention, il faut que la mémoire pointée par `dest` soit assez grande pour stocker les `taille` octets. Sinon, corruption mémoire ou `segfault`.

```
char dest[100] = "Bonjour"; // 7 caractères
char *src = "Albert Victor";// Albert: 6 car.  Victor: 6 car.
printf("%s\n",dest);
memcpy(dest + 7,src + 6,7); // ok, car dest peut contenir 100 octets
printf("%s\n",dest);
```

```
Bonjour
Bonjour Victor
```

Fonctions `mem...`

```
void * memset (void * ptr, int valeur, size_t taille );
```

Affecte `valeur` à `taille` octets en commençant à `ptr`.

```
char dest[100];
memset(dest,'A',100);
dest[5] = '\0'; // <- pour avoir une C-chaîne
printf("%s\n",dest);
```

```
AAAAA
```

Fonctions `mem...`

```
void * memset (void * ptr, int valeur, size_t taille );
```

Affecte `valeur` à `taille` octets en commençant à `ptr`.

```
char dest[100];
memset(dest,'A',100);
dest[5] = '\0'; // <- pour avoir une C-chaîne
printf("%s\n",dest);
```

```
AAAAA
```

Utilisation habituelle : initialiser à 0 une structure.

```
struct ma_structure s;
memset(&s,0,sizeof(struct ma_structure));
```

Fonctions `mem...`

```
const char * strstr (const char * str1, const char * str2 );
```

Trouve la première occurence de `str2` dans `str1`. Renvoie `NULL` si la chaîne n'est pas trouvée.

```
char *str1 = "Les chaussettes de l'archiduchesse";
char *str2 = "aus";
printf("%s",strstr(str1,str2));
```

```
aussettes de l'archiduchesse
```

Entrées / sorties

`#include <stdio.h>` permet aussi faire des entrées / sorties en travaillant directement avec des blocs d'octets au lieu de C-chaînes.

Entrées / sorties

`#include <stdio.h>` permet aussi faire des entrées / sorties en travaillant directement avec des blocs d'octets au lieu de C-chaînes.

```
size_t fread (void * ptr, size_t taille, size_t longueur, FILE * f );
```

Lit `taille * longueur` octets dans `f` et les met à `ptr`.

```
int tab[100];
fread(tab,sizeof(int),100,stdin); // <- lit 4 * 100 octets dans tab
```

Entrées / sorties

`#include <stdio.h>` permet aussi faire des entrées / sorties en travaillant directement avec des blocs d'octets au lieu de C-chaînes.

```
size_t fwrite (void * ptr, size_t taille, size_t longueur, FILE * f );
```

Écrit `taille * longueur` octets dans `f` lus à partir de `ptr`.

```
char str[100] = "Bonjour Lucie !";
fwrite(tab + 3,sizeof(char),4,stdout);
```

```
jour
```